Spring 中文文档 3.1

wizardforcel

Published with GitBook

目錄

Spring 中文文档 3.1

Ver 1.6.1（build on 20111007）

南磊 译

电子邮件：nanlei1987@gmail.com 微博：http://weibo.com /nanlei1 987

第一部分 Spring framework 概述

Spring framework 是一个轻量级的解决方案，在构建一站式企业级应用程序上有很大的 潜能。Spring 是模块化的，允许你使用仅需要的部分，而不需要引入其余部分。你可以使用 IoC 容器，和 Struts 一起使用，而且你也可以仅仅使用 Hibernate 整合代码或者是 JDBC 抽象 层。Spring framework 支持声明式的事务管理，通过 RMI 或 Web Service 远程访问业务逻辑 代码，并且提供多种持久化数据的选择。它提供饿了一个全功能的 MVC 框架，允许你显式 地整合 AOP 到软件中。

Spring 被设计成非侵入式的，也就是说你的业务逻辑代码通常是不会对 Spring 框架本身 产生依赖的。在你的整合层面（比如数据访问层），一些依赖于数据访问技术和 Spring 的类 库是会存在的。但是，也很容易将这些依赖从你剩余的代码中分离出来。

本文档是 Spring 框架特性的参考指南。如果你有任何想法，建议或是对本文档的疑问， 请发送到用户邮件列表中或者是在线论坛中，论坛地址是 http://forum.spri ngsource.org。

第 1 章 Spring Framework 介绍

Spring Framework 是一个 Java 平台，它提供了对用 Java 语言开发应用程序的一种广泛的基础支持。Spring 本身来控制这个基础，那么你就可以集中精力于应用程序的开发了。

Spring 允许你从“普通 Java 对象（POJO）”来构建应用程序，并且将非侵入地企业级服 务应用于 POJO 中。这种能力不仅适用于 Java SE 编程模型，也适用于全部或部分的 Java EE。

作为应用程序的开发人员，下面就是可以使用 Spring 平台所含优点的例子：

编写 Java 方法来执行数据库事务而不需要处理相关的事务 API。

编写本地的 Java 方法来访问远程程序而不需要处理远程访问 API。

编写本地的 Java 方法来执行管理操作而不需要处理 JMX 的 API。

编写本地的 Java 方法来处理消息操作而不需要处理 JMS 的 API。

1.1 依赖注入和控制反转

背景

“问题是，[它们]反向控制哪一方面？”，2004 年，Martin Fowler 在他个人站点提出 了这个关于控制反转（IoC）的问题。Fowler 建议重命名这个原则，使得它更好地自我解 释，同时提出了依赖注入。

要 深 入 了 解 IoC 和 DI ， 可 以 参 考 Fowler 的 文 章 ， 地 址 是 ：

http://mart inf owler.com/artic les/injection.html

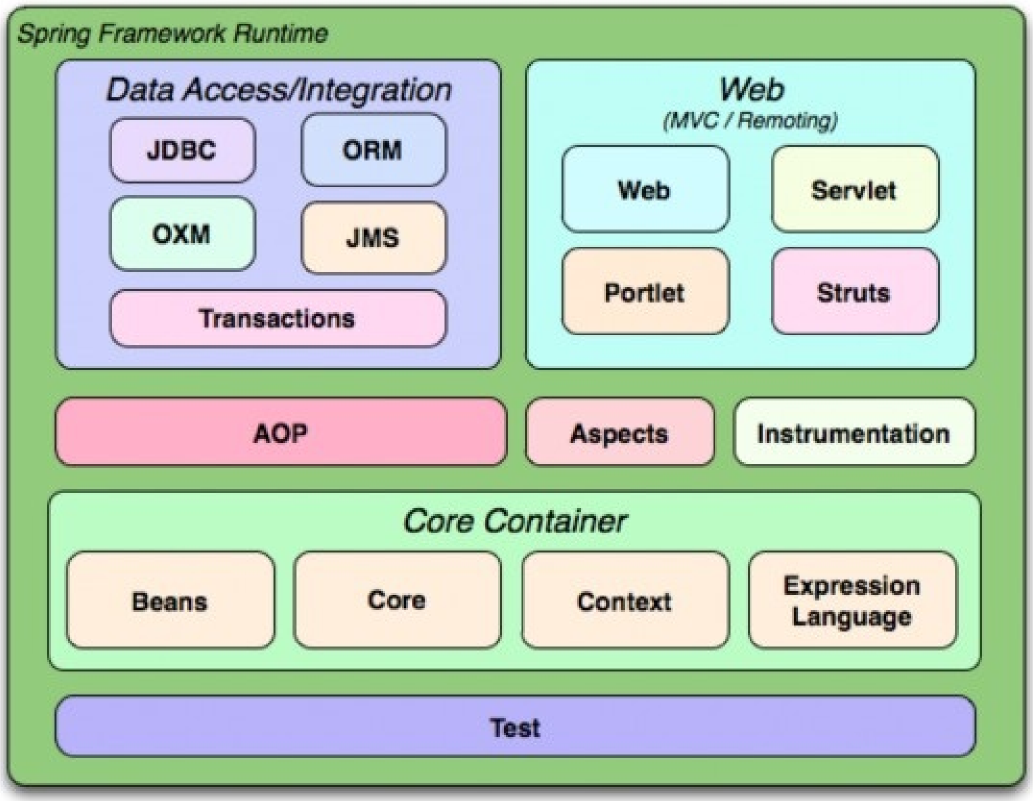
Java 应用程序 -- 一个宽松的术语，囊括了从被限制的 applet 到 n 层服务器端的企业级 应用程序的全部 – 典型的应用是，包含了组成独特应用程序的合作对象。那么在应用程序 中的这些对象就会有相互依赖关系。

尽管 Java 平台提供了丰富的应用程序开发功能，但是它也缺乏组织基本模块到整体的 方式，而是把这个任务留给了系统架构师和开发人员去解决。也就是说，你可以设计如工厂，抽象工厂，构建者，装饰者和服务定位器等设计模式来组合各个类，以及构成该应用程序的对象的实例。然而，这些模式都是最简单的：最佳的做法是给定一个名称，并且描述这个模式做了些什么，在哪里可以应用它，它所强调的问题是什么等等。模式可以使得你必须自己实现的最佳实践形式化。

Spring Framework 的控制反转（Inversion of Control，IoC）组件提供了组合不同的组件到 完整可用的应用程序的形式化方法。Spring Framework 编写了形式化的设计模式作为顶级对象，你可以用来整合到你自己的应用程序中。很多组织和研究机构使用 Spring Framework 的这个方式来设计健壮的，可维护的应用程序。

1.2 模块

Spring Framework 包含了很多特性，并且组织成 20 个模块。这些模块分为核心容器， 数据访问/整合，Web，AOP（Aspect Oriented Programming，面向切面编程），基础组件和测试，在下图中来展示这些模块。



Spring 框架概要

1.2.1 核心容器

核心容器（4.1 节）包含了核心（Core），Bean 组件（Beans），上下文（Context）和表达式语言（Expression Language）模块。

核心和 Bean（4.1 节）模块提供了框架部分的基础，包含 IoC 和依赖注入特性。

BeanFactory 是工厂模式的精密实现。它去掉了编程实现单例的需要，并允许你解除配置 信息，以及实际程序逻辑的特定依赖之间的耦合。

上下文（4.14 节）模块构建了由核心和 Bean（4.1 节）模块提供的坚实基础：它可以让 你以框架中的风格来访问对象，这和 JNDI 的注册是相似的。上下文模块集成了来自 Bean 模块的特性，并且加入了对国际化（比如使用资源束）的支持，事件传播，资源加载和 Servlet容器显式创建上下文。上下文模块也支持 Java EE 特性，比如 EJB，JMX 和基本的远程调用。

ApplicationContext 接口是上下文模块的焦点。

表达式语言（第 7 章）模块提供了强大的表达式语言，在运行时查询和操作对象图。这 是 JSP 2.1 规范中的统一表达式语言（Unified EL）的一个扩展。该表达式语言支持设置和获取属性值，属性定义，方法调用，访问数组，集合以及索引的上下文。支持逻辑和数字运算，命名变量。还支持从 Spring 的 IoC 容器中以名称来获取对象。它也支持 list 的投影和选择操作，还有普通的 list 聚集。

1.2.2 数据访问/整合

数据访问/整合层由 JDBC，ORM，OXM，JMS 和事务模块组成。

JDBC（13.1 节）模块提供了 JDBC 抽象层，它移除了冗长的 JDBC 编码，但解析了数据库提供商定义的特定错误代码。

ORM（14.1 节）模块提供了对流行的对象-实体映射 API 的整合层，包含 JPA（14.5 节），JDO（14.4 节），Hibernate（14.3 节）和 iBatis（14.6 节）。使用 ORM 包你就可以使用全部的O/R-映射框架并联合其它 Spring 提供的特性，比如之前所提到的简单声明式的事务管理特性。

OXM（第 15 章）模块提供了支持 JAXB，Castor，XMLBeans，JiBX，以及 Xstream 对对象/XML 映射实现的抽象层。

Java 消息服务（JMS，第 22 章）模块包含生成和处理消息的特性。

事务（第 11 章）模块支持对实现特定接口的类和所有 POJO（普通 Java 对象）的编程 式和声明式的事务管理。

1.2.3 Web

Web 层由 Web，Web-Servlet，Web-Struts 和 Web-Portlet 模块组成。

Spring 的 Web 模块提供了基本的面向 Web 整合的特性，比如文件上传功能，IoC 容器 的初始化使用了 Servlet 的监听器和变形的 Web 应用上下文。它还包含了和 Web 相关的 Spring 远程调用支持的部分。

Web-Servlet 模块包含了 Spring 对 Web 应用的模型-视图-控制器（MVC，16.1 节）模式 的实现。Spring 的 MVC 框架提供了一个在领域模型代码和 Web 表单之间的整洁分离，并且 整合了其它所有 Spring 框架的特性。

Web-Struts 模块包含了使用 Spring 应用程序整合经典的 Strtus Web 层的支持类。要注意 这个支持从 Spring 3.0 开始就废弃了。那么可以考虑迁移应用程序到 Struts 2.0 和 Spring 的整合或者是单纯的 Spring MVC 方案。

Web-Portlet 模块提供用于 portlet 环境和 Web-Servlet 模块功能镜像的 MVC 实现。

1.2.4 AOP 和基础组件

Spring 的 AOP（8.1 节）模块提供了 AOP 联盟-允许的的面向切面的编程实现，允许你定 义如方法-拦截器和横切点来整洁地解耦应该被分离的功能实现代码。使用源代码级的元数 据功能，也可以混合行为信息到代码中，这个方式和.NET 的属性很相似。

分离的 Aspects 模块提供对 AspectJ 的整合。

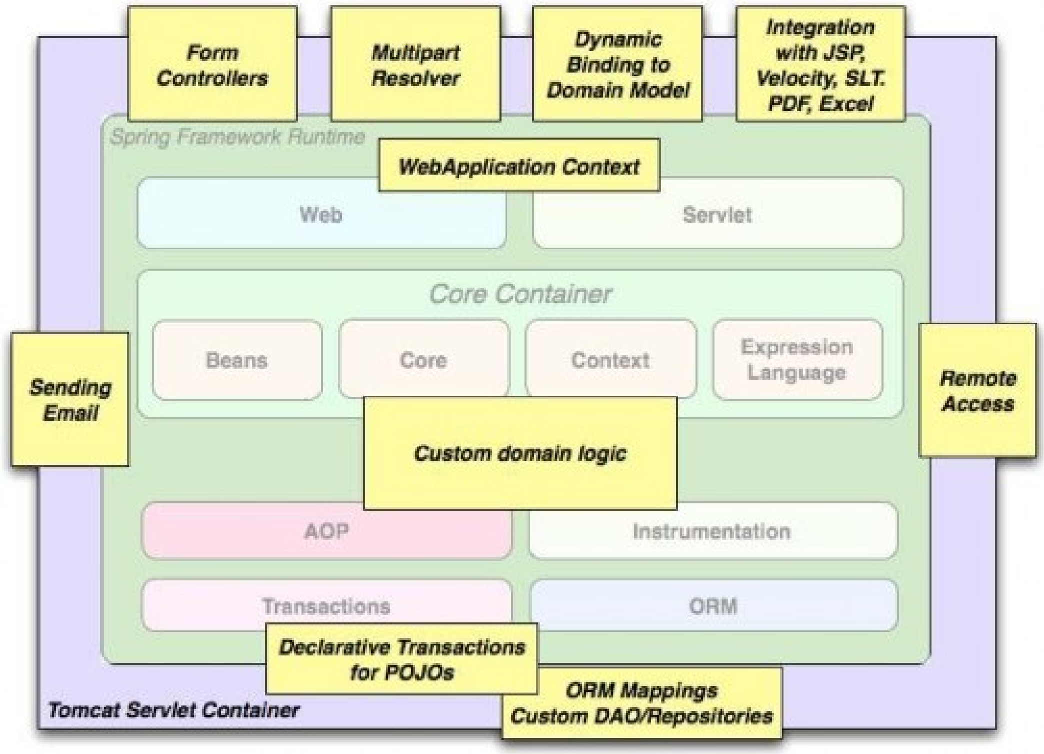
基础组件模块提供了对类的基础支持，还有类加载器的实现来用于特定的应用服务器。

1.2.5 测试

测试模块支持 Spring 组件，Junit 或 Test NG 的测试。它提供了 Spring 应用上下文的一致 加载并缓存这些上下文内容。它也提供 mock 对象，你可以用来孤立地测试代码。

1.3 使用方案

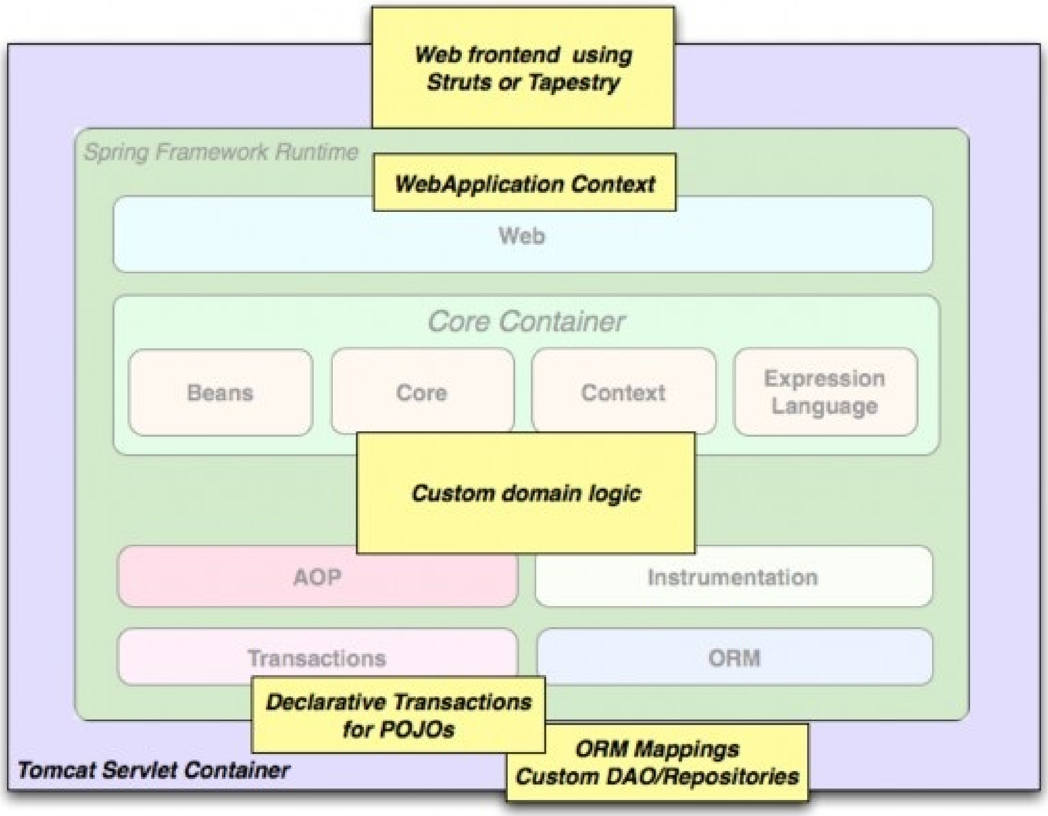
前面描述的模块使得 Spring 可以在很多方案中作为业务逻辑实现的选择，从 applet 到 使用了 Spring 的事务管理功能和 Web 框架整合，功能完善的企业级应用程序。



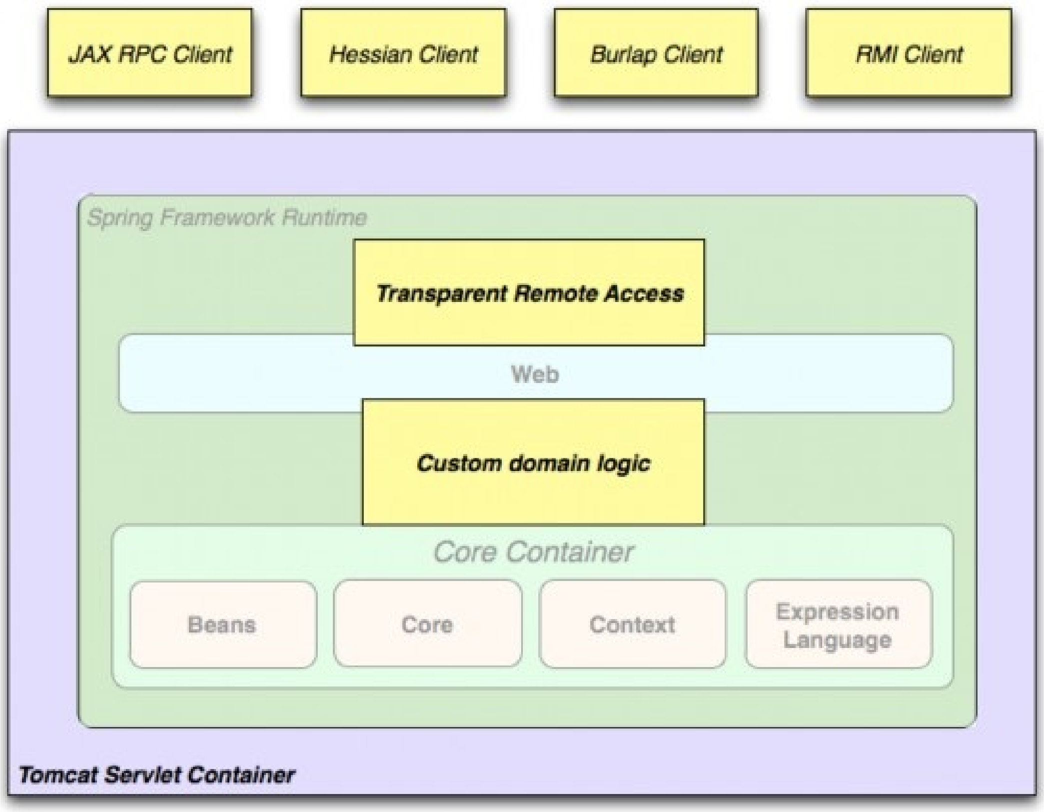
典型地功能完善的 Spring Web 应用

Spring 的声明式事务管理特性（11.5 节）使得 Web 应用程序可以全部事务化，就好像

使用了 EJB 容器管理的事务。全部的自定义业务逻辑可以使用简单的 POJO 来实现并且被 Spring 的 IoC 容器来管理。其它的服务包含对发送邮件的支持，验证对 Web 层独立，这可以让你选择在哪里执行验证规则。Spring 的 ORM 支持对 JPA，Hibernate,JDO 和 iBatis 进行了整合；比如，当使用 Hibernate 时，你可以继续使用已有的映射文件和标准的 Hibernate 的 SessionFactory 配置 。表 单控 制器 无缝 地整 合了 Web 层 和领 域模 型， 也去 除了 ActionForm 或其它为领域模型转换 HTTP 参数的类的需要。

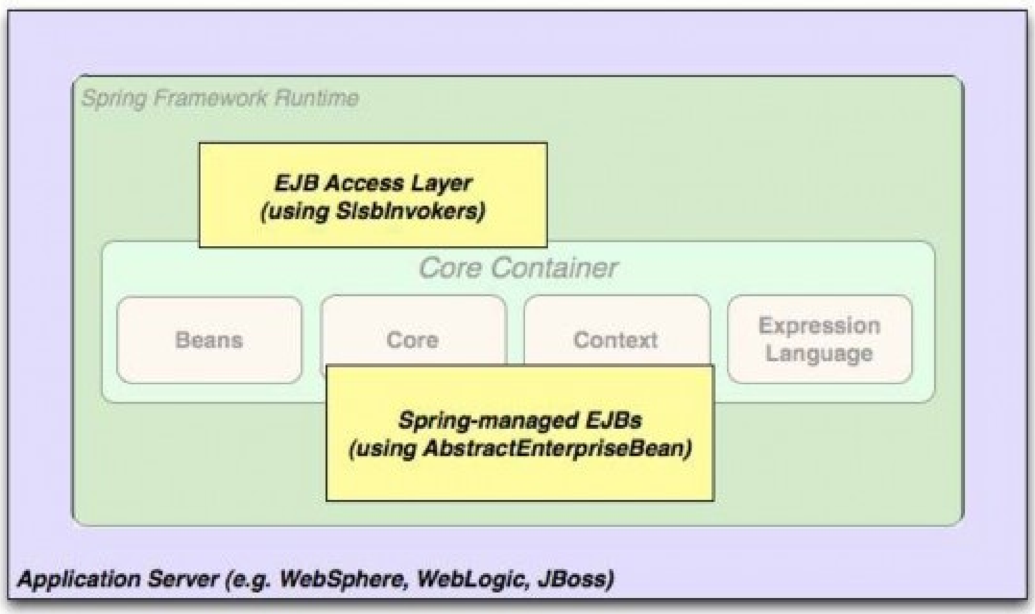


使用了第三方 Web 框架的 Spring 中间层 有时，开发和运行环境并不允许你完全转换到一个不同的框架中。Spring Framework 不强制你使用其中的部分；它并不是一个所有或没有的方案。已有使用了 WebWork，Struts， Tapestry 或其它 UI 框架构建的前端应用程序也可以使用基于 Spring 的中间层来进行整合， 这 就 允 许 你 去 使用 Spring 的 事 务 特 性 。 你 仅 仅 需 要 做 的 是 给 业 务 逻 辑 装 上 ApplicationContext，对整合的 Web 层使用 WebApplicationContext。



远程调用使用方案

当你需要通过 Web Service 访问已有的代码时，你可以使用 Spring 的 Hessian-， Burlap-，Rmi-或 JaxPpcProxyFactory 类。对已有的应用程序开启远程访问并不困难。



EJB – 包装已有的 POJO

Spring Framework 也提供了对企业级 Java Bean（EJB，译者注）的访问和抽象层（第 21 章），这就可以重用已有的 POJO，可以扩展、包装它们到无状态会话 bean，不安全的 Web 应用程序可能也需要声明式安全。

1.3.1 依赖管理和命名规约

依赖管理和依赖注入是不同的概念。要在应用程序中添加 Spring 的优雅特性（比如依 赖注入），你需要放置所需的类库（jar 文件），并且添加到运行时环境的类路径中，通常在 编译时也是需要这些类库文件的。这些依赖不是注入的虚拟组件，而是文件系统（通常来说 是这样）上的物理资源。依赖管理的过程包括定位资源，存储它们，并将它们添加到类路径 中。依赖可以是直接的（比如应用程序在运行时需要 Spring），或者是间接的（比如应用程 序需要的 commons-dbcp 组件还依赖于 commons-pool 组件）。间接的依赖也被认为是“过 度的”，而且那些依赖本身就难以识别和管理。

如果你决定使用 Spring，那么你需要获得 Spring 的 jar 包，其中要包括你所需要使用的 Spring 的模块。为了使用的便捷，Spring 被打包成各个模块的集合，尽可能地分离其中的相互依赖，那么如果你不想编写 Web 应用程序，就不需要添加 spring-web 模块的 jar 包。要参考 Spring 模 块 的 库 ， 本 指 南 使 用 了 一 个 可以 速记 的 命 名 规 约 ， spring- 或者 *spring*-.jar ，这里的“ * ” 代 表 了 各 模 块 的 短 名 称 （ 比 如 ， spring-core ， spring-webmvc，spring-jms 等）。实际上 jar 文件命名或许就是这种形式的（参考下 面的示例），但也有可能不是这种情况，通常它的文件名中还包含一个版本号（比如， spring-core-3.0.0.RELEASE.jar）。

通常，Spring 在四个不同的地方发布组件：

- 在社区下载点 http://www.springsource.org/dow nloads/comm unity。在这儿你可以找到所 有的 Spring jar 包，它们被压缩到一个 zip 文件中，可以自由下载。这里 jar 包的命名从 3.0 版本开始，都以 org.springframework.*-.jar 格式进行。
- Maven 的中央库，也是 Maven 默认检索的资源库，它并不会检索特殊的配置来使用。很多 Spring 所依赖的常用类库也可以从 Maven 的中央库中获得，同时 Spring 社区的绝大多数用户都使用 Maven 作为依赖管理工具，这对于他们来说是很方便的。这里的 jar 包的命名规则是 spring-*-.jar 格式的，并且 Maven 里的 groupId 是 org.springframework。
- 企业级资源库（Enterprise Bundle Repository，EBR），这是由 SpringSource 组织运营的， 同时也提供了和 Spring 整合的所有类库。对于所有 Spring 的 jar 包及其依赖，这里也有 Maven 和 Ivy 的资源库，同时这里还有很多开发人员在使用 Spring 编写应用程序时能用到的其它大量常用类库。而且，发布的版本，里程碑版本和开发版本也都部署在这里。 这 里 jar 文 件 的 命 名 规 则 和 从 社 区 下 载 的 （ `org.springframework.*-<version>.jar` ）一致，并且所依赖的外部类库（不 是来自 SpringSource 的）也是使用的这种“长命名”的形式，并以 com.springsource 作为前缀。可以参考 FAQ 部分获取更多信息。
- 在 Amazon S3 为开发和里程碑版本发布（最终发布的版本这里也会有）而设置的公共 Maven 资源库。Jar 文件的名称和 Maven 中央库是一样的，那么这里是获取 Spring 开发版本的地方，其它的类库是部署于 Maven 中央库的。 那么，首先你要决定的事情是如

何管理这些依赖：很多人使用自动化的系统，比如 Maven 和 Ivy，但是你也可以手动下载所有的 jar 文件。当使用 Maven 或 Ivy 获取 Spring 时，之后你就需要决定从哪里来获取它们。通常来说，如果你关注 OSGi 的话，那么就使用 EBR，因为 它对所有的 Spring 依赖兼容 OSGi，比如 Hibernate 和 Freemarker。如果对 OSGi 不感兴趣， 那么使用哪个下载都是可以的，但是他们也各有利弊。通常来讲，为你的项目选择一个或者备用的一个；但是请不要混用。因为相对于 Maven 中央库而言，EBR 组件非常需要一个不同的命名规则，那么这就特别重要了。

表 1.1 Maven 中央库和 SpringSource EBR 资源库的比较

特性	**Maven 中央库**	**EBR**
OSGi 的兼容	不明确	是
组件数量	成千上万；所有种类	几百个；是 Spring 整合中会用到的
一致的命名规约	没有	有
命名规约：GroupId	相异。新组件通常使用域名，比如 org.slf4j。老组件通常仅仅使用组件名，比 如 log4j。	原 始 的 域 名 或 主 包 的 根 路 径 ，比 如 org.springframework
命名规约：ArtifactId	相异。通常是项目或模块名，使用连字符“-”分隔， 比如 spring-core，log4j。	捆绑符号名称，从主包的路径分离，比如 org.springframework.beans。如果 jar 需要修补以保证兼容 OSGi，那么就附加 com.springsource， 比 如 com.springsource.org.apache.log4j
命名规约：Version	相异。很多新的组件使用 m.m.m 或 m.m.m.X（m 是数字，X 是文本）。老的使用 m.m。有一些也不是。顺序是确定的但通常并不 可靠，所以不是严格的可 靠。	OSGi 版本数字 m.m.m.X，比如 3.0.0.RC3。 文本标识符使用相同的数字值规定版本 的字母顺序。
发布	通常是自动通过 rsync 或源码控制更新。项目作者 可以上传独立的 jar 文件到 JIRA。	手动（由 SpringSource 控制的 JIRA）
质量保证	根据政策。精确度是作者的责任。	宽泛的 OSGi 清单，Maven POM 和 Ivy 元数据。QA 由 Spring 团队执行。
主机	Contegix 提供。由 Sonatype 和一些镜像构成。	由 SpringSource 的 S3 构成
搜索工具	很多	http://www.springsource.com/repository
和 SpringSource 工具整合	通过和 Maven 依赖管理的 STS 来整合	通过 Maven， Roo， CloudFoundry 的 STS 进行宽泛整合

1.3.1.1 Spring 依赖和基于 Spring

尽管 Spring 提供对整合的支持，以及对大量企业级应用及其外部工具的支持，那么它 也有心保持它的强制依赖在一个绝对小的数目上：你不需要为了简单的使用去定位并下载

（甚至是自动地）大量的 jar 来使用 Spring。对于基本的依赖注入那只需要一个必须的外部依赖，就是日志（可以参考下面关于日志的深入介绍）。

下面，我们来概述一下配置一个基于 Spring 的应用程序所需的基本配置，首先使用

Maven 和 Ivy。在所有的示例中，如果有哪一点不清楚，可以参考你所使用的依赖管理系统的相关文档，或者参考一些示例代码 – Spring 本身在构建时使用了 Ivy 来管理依赖，而我们

大多数的示例却使用 Maven。

1.3.1.2 Maven 依赖管理

如果你正使用 Maven 来进行依赖管理，那么你就不需要明确地提供日志依赖。比如， 要为应用程序配置创建应用上下文并使用依赖注入特性，那么，你的 Maven 依赖配置文件 可以是这样的：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

就是这么简单。要注意的是，如果你在编译代码时不需要使用 Spring 的 API，那么 scope

元素可以声明为 runtime，这就可以用于典型的依赖注入用例。 在上面的示例中，我们使用了 Maven 中央库中的命名规约，那么它就可以在 Maven 中

央库或 SpringSource S3 的 Maven 资源库中起作用。要使用 S3 的 Maven 资源库（比如里程碑

版本或开发快照）版本，那就需要在 Maven 的配置文件中明确地指定资源库的位置。对于 完整发布版，配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.release</id>
        <url>http://maven.springframework.org/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

对于里程碑版本配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.milestone</id>
        <url>http://maven.springframework.org/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

对于开发快照版本配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.snapshot</id>
        <url>http://maven.springframework.org/snapshot/</url>
        <snapshots><enabled>true</enabled>< /snapshots>
    </repository>
</repositories>
```

要使用 SrpingSource 的 EBR 的话，你还需要一个对依赖不同的命名规约。名称通常很容 易去猜测，比如这个示例中，就是：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>org.springframework.context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

你也可能需要明确地声明资源库的位置（仅仅 URL 是重要的）：

```
<repositories>
    <repository>
        <id>com.springsource.repository.bundles.release</id>
        <url>http://repository.springsource.com/maven/bundles/release/</url>
    </repository>
</repositories>
```

如果你要手动地去管理依赖，在上面这个声明的资源库 URL 是不可以浏览的，但是它 也有一个用户界面，地址是 http://www.springsource.com/repositor y，这可以用来搜索和下 载依赖。它也有 Maven 和 Ivy 配置的代码片段，你可以复制下来并粘贴到你所使用的工具中， 很方便使用。

1.3.1.3 Ivy 依赖管理

如果你使用 Ivy 来管理依赖，那么有一些简单的命名规约和配置选项。

要配置 Ivy 定位到 SpringSource EBR 中，需要添加如下的解析器元素到你的配置文件

ivysettings.xml 中：

```
<resolvers>
    <url name="com.springsource.repository.bundles.release">
        <ivy pattern="http://repository.springsource.com/ivy/bundles/release
        /
        [organisation]/[module]/[revision]/[artifact] -[revision].[ext]"
        />
        <artifact pattern="http://repository.springsource.com/ivy/bundles/release
        /
        [organisation]/[module]/[revision]/ [artifact]-[revision].[ext]"
        />
    </url>
    <url name="com.springsource.repository.bundles.external">
        <ivy pattern="http://repository.springsource.com/ivy/bundles/externa l/
        [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
        />
        <artifact pattern="http://repository.springsource.com/ivy/bundles/externa l/
        [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
        />
    </url>
</resolvers>
```

上面的 XML 并不是合法的格式，因为行太长了 – 如果你要复制粘贴，那么要移除中部

url 模式部分结尾额外的行。（本文档中已经去除）

一旦 Ivy 配置好去查看 EBR，那么添加依赖就很简单了。只要拉起在资源库浏览器中的 捆绑详细信息页面，你就会发现为你准备好的 Ivy 代码片段，那么就可以将它包含到你的依

赖部分中了。比如（在 ivy.xml 中）：

```
<dependency org="org.springframework" name="org.springframework.core" rev="3.0.0.RELEASE"
conf="compile->runtime"/>
```

1.3.2 日志

对于 Spring 来说，日志是一个非常重要的依赖，因为 a)这是唯一强制的外部依赖，b) 开发人员都会想看到他们所使用的工具的一些输出内容，而且 c)Spring 整合了多种实用工具， 它们都会选择一种日志依赖包。应用程序开发人员的目标之一就是在核心位置对整个应用程 序有一个统一的日志配置，包括对所有的外部组件。因为日志框架有多种选择，那么这就可 能有些难以确定了。

Spring 中强制的日志依赖包是 Jakarta 的 Commons Logging API（JCL）。我们对 Spring 的编译是基于 JCL 的，而且我们使扩展了 Spring Framework 的类对 JCL 包的 Log 对象都是可见 的。对于用户来说，所有 Spring 的版本都使用相同的日志包也是很重要的：因为保留了向后兼容的特性，那么迁移是很容易进行的，同理，扩展 Spring 的应用程序也是这样。我们

这样做就是使得 Spring 中的模块明确地使用基于 commons-logging（JCL 的典型实现）的 日志实现，在编译时也使得其它模块都基于这个日志包。如果你正在使用 Maven 的话，同 时想知道 在 哪儿 获 取 到的 commons-logging 依赖 ， 那就是从 Spring 中被称作是 spring-core 的核心模块中获取的。

关于 commons-logging 比较好的做法是你不需要做其它的步骤就可以使应用程序运 行起来。它有一个运行时的查找算法，在我们都知道的类路径下寻找其它的日志框架，并且

使用 Spring 认为是比较合适的（或者告诉 Spring 你需要使用的具体是哪一个）一个。如果

没有找到可用的，那么你会得到一个来自 JDK（java.util.logging 或者简称为 JUL）本身看起来 还不错的日志依赖。那么，你会发现在很多时候，Spring 应用程序运行中会有日志信息打印 到控制台上，这点是很重要的。

1.3.2.1 不使用 Commons Logging

不幸的是，commons-logging 的运行时查找算法对最终用户方便是有些问题的。如 果我们可以让时光倒流并让 Spring 从现在开始作为一个新的项目进行，那么我们会使用不 同的日志依赖。首选的日志依赖可能就是 Java 简单的日志门面（SLF4 J），SLF4J 也被 Spring 的开发人员在他们其它应用程序中的很多工具所使用。

关闭 commons-logging 也很简单：仅仅要确认在运行时，它的包不在类路径下就可 以了。在 Maven 中要去掉这个依赖，因为 Spring 依赖声明的时候会带进来，那么就需要这 么来进行。

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

目前这个应用程序可能就不能运行了，因为在类路径中已经没有了 JCL API 的实现了， 所以要修复这个问题的话，就要提供另外一种日志的实现了。在下一节中，我们来说明如何 提供一个其它的 JCL 的实现，这里，我们使用 SLF4J 作为示例。

1.3.2.2 使用 SLF4J

SLF4J 是一个整洁的日志依赖，在运行时的效率也比 commons-logging 更高，因为

SLF4J 使用了编译时构建，而不是运行时去查找其它整合的日志框架。这也就意味着你可以

更明确地在运行时去做些什么，并且去声明或配置。SLF4J 为很多通用的日志框架提供的绑定，所以通常你可以选择已有的一个，并去绑定配置或管理。

SLF4J 为很多通用日志框架提供绑定，包括 JCL，并且也可以反向进行：桥接其它的日志 框架和 SLF4J 本身。所以如果要在 Spring 中使用 SLF4J，那么就需要使用 SLF4J-JCL 桥来代替

commons-logging 依赖。只要配置好了，那么来自 Spring 的日志调用就会被翻译成调用

SLF4J 的 API 了，如果在应用程序中的其它类库使用了原有的 API，那么会有一个单独的地方 来进行配置和管理的日志。

一个常用的选择就是桥接 Spring 到 SLF4J API，之后提供从 SLF4J 到 Log4J 的明确绑定。你需要提供 4 种依赖（并排除已经存在的 commons-logging 依赖）：桥接工具，SLF4J 的

API，绑定到 Log4J 的工具，还有 Log4J 本身的实现。那么在 Maven 的配置文件中，你就可以这样来做：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

这样，看起来好像很多依赖都需要日志了。情况确实是这样，但 SLF4J 是可选的，而且 它的性能要比含有类加载器问题的 commons-logging 依赖好很多，尤其是如果你使用了 一个严格限制的容器，比如 OSGi 平台。据称那也会有性能优势，因为绑定是编译时进行的， 而不是在运行时。

另外，在 SLF4J 的用户中一个较为常见的选择是使用很少步骤，同时产生更少的依赖， 那也就是直接绑定到 Logback 上。这会去掉很多额外的绑定步骤，因为 Logback 本身直接实 现了 SLF4J ， 这 样 的 话， 你 就仅 需 要两个依赖的类库 ， 而不 原先 的 是四 个 了 (就是 jcl-over-slf4j 和 logback）。如果你也确实那么来做了，你可能还需要从其它外部依 赖（而不是 Spring）中去掉 slf4j-api 的依赖，因为在类路径中只保留一个 API 的一个版本就 行了。

1.3.2.3 使用 Log4J

很多用户出于配置和管理的目的而使用 Log4j 作为日志框架。这样也同样很有效率并且 易于创建，而且，Log4j 也是我们事实上在构建和测试 Spring 时，和运行时环境中使用的日 志框架。Spring 本身也提供一些工具来配置和初始化 Log4j，所以，在某些模块中它也提供 了可选的在编译时对 Log4j 框架的依赖。

要让 Log4j 框架和默认的 JCL 依赖（commons-logging）同时起作用，你所要做的就 是要将 Log4j 的类库 放到 类路 径下 ， 并且 提供 配置 文件 （在类 路径 的根 路径 下放置 log4j.properties 或 log4j.xml 为名的文件）。而对于使用 Maven 的用户来说， 下面 的代码可以是对依赖的声明：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

下面是 log4j.properties 打印到控制台的日志的配置示例：

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j .ConsoleAppender log4j.appender.stdout.layout=org.

log4j.category.org.springframework.beans.factory=DEBUG
```

运行时容器和本地的 JCL

很多用户在容器中运行 Spring 的应用程序，而该容器本身也提供了 JCL 的实现。比如 IBM 的 Webshphere 应用服务器（WAS）就是这种类型的。这通常就会引起问题，而不幸的是没有什么解决的方法；很多情况下仅仅从应用程序中去掉 commons-logging 依赖是不够的。

我们要 清楚 地认 识这一 点： 这个 问题 通常和 JCL 本 身一同 报告 出来， 或 者是和

commons-logging：而不是绑定 commons-logging 到另外的框架（这里，通常是 Log4J）。这就可能会引发问题，因为 commons-logging 改变了它们在运行时环境里，在一些容器 中查找老版本（1.0）的方式，而现在很多人使用的是新版本（1.1）。Spring 不会使用任何不 通用的 JCL API 部分，所以这里不会有问题，但是 Spring 本身或应用程序尝试去进行日志记录，你可以发现绑定到 Log4J 是不起作用的。

这种在 WAS 上的情况，最简单的做法就是颠倒类加载器的层次（IBM 称之为“parent last”），那么就是使应用程序来控制，而不是容器来控制 JCL 的依赖。这种选择并不总是开放的，但是在公共领域中的替代方法也有其它的建议，根据确切的版本和容器的特性集，您所要求的效果可能会有不同。

第二部分 Spring 3 的新特性

第 2 章 Spring 3.0 的新特性和增强

如果你使用 Spring Framework，那么你会看到 Spring 现在主要有两个修订版本：一个是 在 2006 年 12 月发布的 Spring 2.0，还有一个是在 2007 年 11 月发布的 Spring 2.5。而现在是第三个版本 Spring 3.0 了。

Java SE 和 Java EE 支持

Spring Framework 现在完全是基于 Java 5 和 Java 6 版本的。

此外，Spring 还兼容 J2EE 1.4 和 Java EE5，同时也会介绍一些对 Java EE6 的先期支持。

2.1 Java 5

框架的整体代码都已经修订到支持 Java 5 的新特性了，比如泛型，可变参数和其它的语 言方面的改进。我们也尽最大努力来保持代码的向后兼容性。现在我们也有一致的泛型集合 和 Map 的使用，一致的泛型 FactoryBean 使用，还有在 Spring AOP 的 API 中对桥接方法的一致 性 解决方案。泛型 的 上 下文 监 听 器 仅 仅 自 动 接收 特 定 的 事 件 类 型。 所 有 的 比如 TransactionCallback 和 HibernateCallback 回调接口现在也都声明为泛型返回值。总之，Spring 核心代码库都已经为 Java 5 而进行修订和优化了。

Spring 的 TaskExecutor 抽象也已经为和 Java 5 的 java.util.concurrent 的紧密整合而更新了。 我们现在为可调用特性提供顶级的类的支持，还有 ExecutorService 适配器，ThreadFactory 的整合等。这和 JSR-236（Java EE6 的并发工具）尽可能是一致的。此外，我们还提供对使 用新的@Async 注解（或者 EJB 3.1 的@Asynchronous 注解）的异步方法调用的支持。

2.2 改进的文档

Spring 参考文档也有很大的更新，并反映出对 Spring 3.0 的改进和新特性。而每一项努 力都是来保证在文档中没有错误，但是多多少少还会存在一些错误。如果你确实发现了任意 的错字或者是严重的错误，你可以在午餐时间把它们画上圈，然后将这些错误报告给 Spring 的开发团队，你可以在 JIRA 上打开一个问题。

2.3 新的文章和教程

有很多优秀的文章和教程来展示如何开始使用 Spring 3 的新特性。您可以在 Spring 文档页面来阅读它们。

示例代码也已经更新采用了 Spring 3 的新特性。此外，示例代码也已经从源码树中移入到了专用的 SVN 资源库中。SVN 资源库的访问地址是：https://anonsvn.springframework.org/svn/spring-samples/。

因此，这些示例代码就不会跟在 Spring 3 的发布包中了，需要你从上面提到的资源库中分别去下载。而本文档会继续引用一些示例（特别是 Petclinic）来说明各种特性。注意要获取关于 Subversion（简称为 SVN）的更多信息，可以参考 SVN 的项目主页，地址是：http://subversion.apache.org/。

2.4 新的模块组织方式和系统构建方式

框架的模块已经被重新修订了，现在的组织结构是：一个源码分支是一个 jar 文件：

- org.springframework.aop
- org.springframework.beans
- org.springframework.context
- org.springframework.context.support
- org.springframework.expression
- org.springframework.instrument
- org.springframework.jdbc
- org.springframework.jms
- org.springframework.orm
- org.springframework.oxm
- org.springframework.test
- org.springframework.transaction
- org.springframework.web
- org.springframework.web.portlet
- org.springframework.web.servlet
- org.springframework.web.struts

注意：

原来的 Spring.jar 文件包含了几乎所有框架内容，而现在不再提供了。

现在我们使用新的 Spring 构建系统，从熟知的 Spring Web Flow 2.0 中而来，它给了我们 如下的特性：

- 基于 Ivy 的“Spring 构建”系统
- 一致的开发过程
- 一致的依赖管理
- 一致的 OSGi 清单生成

2.5 新特性概述

下面列出的是 Spring 3.0 的新特性，我们在后面章节的详细讲述中会覆盖到这些特性。

- Spring 表达式语言
- IoC 增加/基于对 Java 的 bean 元数据的支持
- 通用的类型转换和字段格式化系统
- 对象转 XML 映射功能（OXM），这是从 Spring Web Service 项目中迁出的
- 全面的 REST 支持
- 增加@MVC
- 声明式的模型验证
- 先期对 Java EE 6 的支持
- 嵌入式数据库的支持

2.5.1 为 Java 5 更新的核心 API

BeanFactory 接口尽可能地返回该类型 bean 的实例：

- `T getBean(Class<T> requiredType)`
- `T getBean(String name, Class<T> requiredType)`
- `Map<String, T> getBeansOfType(Class<T> type)`

现在，Spring 的 TaskExecutor 接口扩展了 java.util.concurrent.Executor

- 扩展的 AsyncTaskExecutor 支持标准的可调用特性

新的基于 Java 5 的转换 API 和 SPI：

- 无状态的 ConversionService 和转换器
- 取代标准 JDK 的 PropertyEditors

泛型 `ApplicationListener<E>`

2.5.2 Spring 表达式语言

Spring 引入了一种表达式语言，这和统一的 EL 在语法上很相似，但是提供了很多新特 性。这种表达式语言可以用于定义基于 XML 和注解的 bean，也可以作为表达式语言的基础， 支持贯穿整个 Spring 框架的组合。这些新功能的详细内容可以在第 7 章 Spring 表达式语言（SpEL）中看到。

Spring 表达式语言被用来为 Spring 社区提供一个单一的，支持良好的表达式语言，可以 用于 Spring 系列的所有产品中。它的语言特性由 Spring 系列产品的所有项目的需求来共同 驱动，包括基于 Eclipse 的 SpringSource Tool Suite（SpringSource 组织开发的工具套件，译者

注）代码完成支持工具的需求。

下面是表达式语言如何被用于配置数据库属性设置的示例代码：

```
<bean class="mycompany.RewardsTestDatabase">
    <property name="databaseName" value="#{systemProperties.databaseName}" />
    <property name="keyGenerator"
        value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

如果你使用注解来配置组件，那么这个功能也是可用的：

```
@Repository
public class RewardsTestDatabase {
    @Value("#{systemProperties.databaseName}")
    public void setDatabaseName(String dbName) { … }
    @Value("#{strategyBean.databaseKeyGenerator}" )
    public void setKeyGenerator(KeyGenerator kg) { … }
}
```

2.5.3 控制反转（IoC）容器

2.5.3.1 基于 Java 的 bean 元数据

从 Java Config 项目中引入的一些核心特性现在也被加入到 Spring Framework 中了。这就 是说下面的注解是直接被支持的。

- @Configuration
- @Bean
- @DependsOn
- @Primary
- @Lazy
- @Import
- @ImportResource
- @Value

下面是一个 Java 类提供基本配置信息的示例，使用了新的 Java Config 特性：

```
package org.example.config;
@Configuration
public class AppConfig {
    private @Value("#{jdbcProperties.url}") String jdbcUrl;
    private @Value("#{jdbcProperties.username}") String username;
    private @Value("#{jdbcProperties.password}") String password;
    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }
    @Bean
    public FooRepository fooRepository() {
        return new HibernateFooRepository(sessionFactory());
    }
    @Bean
    public SessionFactory sessionFactory() {
        // 装配一个session factory
        AnnotationSessionFactoryBean asFactoryBean =
        new AnnotationSessionFactoryBean(); asFactoryBean.setDataSource(dataSource());
        // 其它配置
        return asFactoryBean.getObject();
    }
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(jdbcUrl, username, password);
    }
}
```

要让它发挥作用，你需要在应用上下文的 XML 文件中添加如下的组件扫描项。

```
<context:component-scan base-package="org.example.config"/>
<util:properties id="jdbcProperties" location="classpath:org/example/config/jdbc.properti
```

或者你可以直接使用 AnnotationConfigApplicationContext 来启动被 @Configuration 注解的类：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig. class);
    FooService fooService = ctx.getBean(FooService.class); fooService.doStuff();
}
```

参考 4.12.2 节“使用 AnnotationConfigApplicationContext 实例化 Spring 容器”来参考关 于 AnnotationConfigApplicationContext 的全部内容。

2.5.3.2 使用组件定义 bean 的元数据

@Bean 注解的方法也可以支持内部的 Spring 组件。它们贡献工厂 bean 的定义到容器中。参考 4.10.4 节“使用组件定义 bean 的元数据”来获取更多信息。

2.5.4 通用的类型转换系统和字段格式化系统

通用的类型转换系统（参考 6.5 节）已经引入了。系统现在被 SpEL 使用来进行类型转 换，当绑定 bean 的属性值时也会被 Spring 容器和数据绑定器使用。

此外，格式化（参考 6.6 节）SPI 也被引入了，来格式化字段值。这个 SPI 对 JavaBean 的 PropertyEditors 提供了简单的，更强壮替代，在如 Spring MVC 的客户端环境中来使用。

2.5.5 数据层

现在，对象到 XML 映射功能（OXM）被从 Spring Web Service 项目中移到 Spring Framework 的核心中。这个功能可以在 org.springframework.oxm 包下找到。关于使用 OXM 模块 的更多信息可以在第 15 章使用 O/X 映射器编组 XML 找到。

2.5.6 Web 层

对于 Web 层来说，最令人兴奋的新特性是对构建 RESTful Web Service 和 Web 应用程序 的支持。也有一些新的注解可以用于任意的 Web 应用程序。

2.5.6.1 全面的 REST 支持

对构建 RESTful 应用程序服务器端的支持已经作为已有的注解驱动的 MVC web 框架而提 供了。客户端的支持由 RestTemplate 类来提供，和其 它的模板类是相 似的，比如 JdbcTemplate 和 JmsTemplate。服务器和客户端两者的 REST 功能都使用 HttpConverter 来在对象和它们在 HTTP 请求和响应代表之间方便的转换。

MarshallingHttpMessageConverter 使用之前提到的对象到 XML 映射功能。 可以参考 MVC（第 16 章）和 RestTemplate（20.9.1 节）部分获取更多信息。

2.5.6.2 @MVC 的增加

mvc 命名空间被引入来大大简化 Spring MVC 的配置。

其它如 @CookieValue 和 @RequestHeader 的 注 解 也 被 加 入 了 。 参 考 使 用

@CookieVlue 注解映射 cookie 值（16.3.3.9 节）和使用@RequestHeader 注解映射请求头部属 性（16.3.3.10 节）来获取更多信息。

2.5.7 声明式的模型验证

一些验证增强（6.7 节），包括 JSR303 支持，使用 Hibernate 校验器作为默认提供者。

2.5.8 先期对 Java EE 6 的支持

通过使用新的@Async 注解（或 EJB 3.1 的@Asynchronous 注解）我们提供异步方法调用。

JSR 303，JSF 2.0，JPA 2.0 等

2.5.9 嵌入式数据库的支持

现在也提供了对嵌入式 Java 数据库引擎（13.8 节）的方便支持，包括 HSQL，H2 和 Derby。

第 3 章 Spring 3.1 的新特性和增强

在 Spring 3.0 引入的支持之上构建，Spring 3.1 现在还在开发中，目前 Spring 3.1 M2 才刚刚发布。

3.1 新特性概述

（该部分内容待 Spring 3.1 GA 发布后翻译）

第三部分 核心技术

参考文档的这一部分涵盖了 Spring Framework 中不可或缺的技术。 这些内容最主要的是 Spring Framework 的控制反转（IoC）容器。Spring Framework 的 IoC

容器的完全使用是紧跟其后的 Spring 的面向切面编程（AOP）技术的完全覆盖。Spring Framework 有它自己的 AOP 框架，在概念上很容易去理解，在 Java 企业级编程中，它成功地解决了 80%的 AOP 需求的功能点。

也提供了涵盖的 Spring 和 AspectJ（目前最丰富的 - 在功能方面 – 当然是在 Java 企业 级空间中最成熟的 AOP 实现）的整合。

最终，通过测试驱动开发（test-driven-development，TDD）的软件开发方法，也是 Spring 团队所主张的，所以 Spring 对整合测试的支持也涵盖到了（沿袭单元测试的最佳实践）。Spring 团队也发现了 IoC 的正确使用，当然，这会让单元和集成测试更容易（setter 方法的存在和 类的适当的构造方法可以使得它们很容易的在测试时连接在一起，而不需要设立服务定位器注册和诸如此类的方法）。这章专门的测试又往说服你。

- 第 4 章，IoC 容器
- 第 5 章，资源
- 第 6 章，验证，数据绑定和类型转换
- 第 7 章，Spring 表达式语言（SpEL）
- 第 8 章，使用 Spring 进行面向切面编程
- 第 9 章，Spring 的 AOP API
- 第 10 章，测试

第 4 章 IoC 容器

4.1 Spring IoC 容器和 bean 的介绍

本章涵盖了 Spring Framework 的控制反转容器（IoC）[参考 1.1 节的背景]原则的实现。 IoC 也被称为是依赖注入（DI）。这是一个对象定义它们依赖的过程，也就是说，它们使用的 其它对象，仅仅通过构造方法参数，工厂方法参数或在对象被创建后的实例上设置的属性， 亦或者是从工厂方法返回的参数。之后容器在它创建 bean 的时候注入那些依赖。这个过程 是根本上的反向，因此名称是控制反转（IoC），bean 本身控制实例化或直接地使用类的构 造来定位它的依赖，或者是如服务定位器模式的机制。

org.springframework.beans 和 org.springframework.context 包是 Spring Framework 的 IoC 容器的基础。BeanFactory 接口提供高级的配置机制，可以管理任意类 型的对象。ApplicationContext 是 BeanFactory 的子接口。它添加了和 Spring 的 AOP 特性很简便的整合；消息资源处理（用于国际化 i18n）， 事件发布；应用层特定的上下文， 比如用于 Web 应用程序的 WebApplicationContext。

总之，BeanFactory 提供了配置框架和基本功能，而 ApplicationContext 添加 了更多企业级开发特定的功能。ApplicationContext 是 BeanFactory 完整的超集， 专门用于本章，来描述 Spring 的 IoC 容器。对于使用 BeanFactory 而不是 ApplicationContext 的更多信息，可以参考 4.15 节“BeanFactory”。

在 Spring 中，对象构成应用程序的骨感，它们是由 Spring 的 IoC 容器管理的，并被称为 bean。一个 bean 就是一个实例化并组装的对象，由 Spring 的 IoC 容器来管理。否则，bean 就是应用程序中众多对象之一。Bean 和它们之间的依赖，反射出由容器使用的配置元数据。

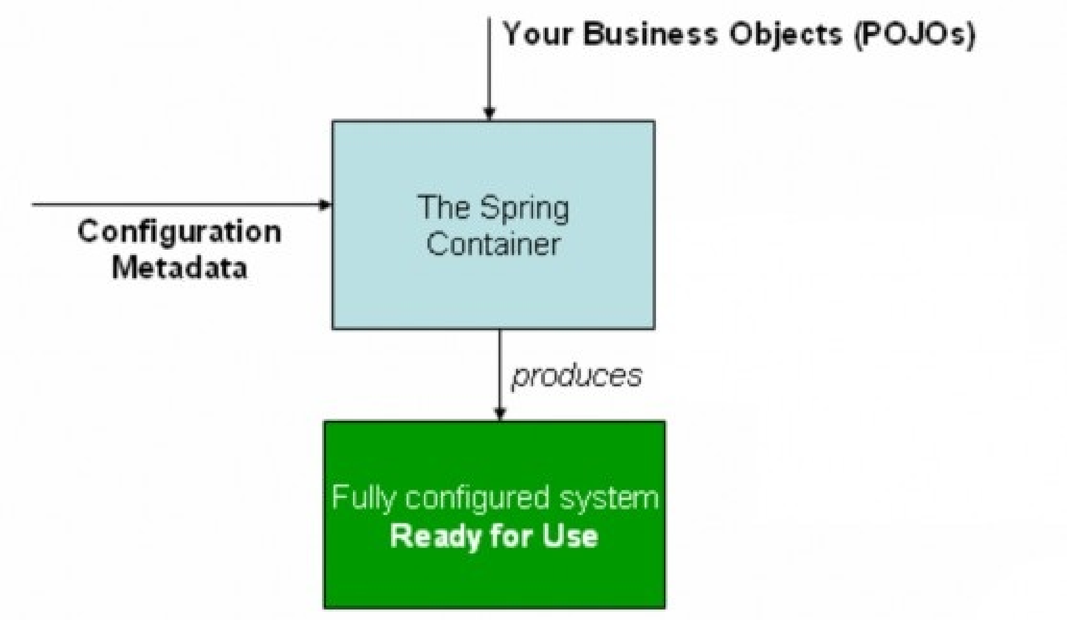
4.2 容器概述

org.springframework.context.ApplicationContext 接口代表了 Spring 的 IoC 容器，负责实例化，配置和装配上述的 bean。容器获得指示来实例化某对象，配置并装 配，这都是通过读取配置元数据实现的。配置元数据在 XML 中，Java 注解或 Java 代码中来 表示。它允许你表达编写应用程序的对象，还有对象间丰富的相互依存的关系。

ApplicationContext 接口的一些实现使用 Spring 开箱的支持。在独立的应用程序 中， 通 常 是 来 创 建 ClassPathXmlApplicationContext 或 FileSystemXmlApplicationContext 的实例。XML 是定义配置元数据的传统格式，你 可以指示容器使用 Java 的注解或是代码作为元数据的格式，提供少量的 XML 配置声明来开 启对这些额外的元数据格式的支持。

在很多应用场景中，明确的用户代码不需要实例化一个或者多个 Spring IoC 容器的实例。 比如，在 Web 应用场景中，在应用程序的 web.xml 中简单的八（左右）行样板 J2EE 描述 符 XML 文件通常就够了（参考 4.14.4 节“对 Web 应用程序方便的应用上下文实例化”）。如 果你正使用 SpringSource 的工具套件，Eclipse 支持的开发环境或者是 Spring ROO 这样的样板 配置，就可以容易地被创建，点几下鼠标或按键就可以了。

下图是 Spring 如何工作的高级别视图。你的应用程序类联合配置元数据，所以在 ApplicationContext 被创建和实例化后，就得到了一个完全配置的可执行系统或程序。



Spring 的 IoC 容器

4.2.1 配置元数据

正如上图所示，Spring 的 IoC 容器处理配置元数据的一种形式；这个配置元数据代表了 你作为应用开发人员是如何告诉 Spring 容器在你的应用程序中来实例化，配置并装配对象 的。

配置元数据传统上是以直观的 XML 格式提供的，这是本章的大部分内容使用它来传达 Spring IoC 容器关键概念和功能。

> 注意
>
> 基于 XML 的元数据并不是唯一的配置元数据格式。这种配置元数据真正写入时，Spring 的 IoC 容器本身和这种格式完全脱钩。

关于 Spring 容器使用元数据格式的信息，可以参考：

基于注解的配置（4.9 节）：Spring 2.5 引入了对基于注解元数据的支持。

基于 Java 的配置（4.12 节）：从 Spring 3.0 开始，很多由 Spring JavaConfig 项目提供的特性称为 Spring Framework 的核心。因此你可以在应用程序外部来定义 bean，使用 Java 代码而不是 XML 文件。要使用这些新的特性，请参考@Configuration，@Bean，

@Import 和@DependsOn 注解

Spring 配置典型的是最少由一个容器必须管理的 bean 定义构成。基于 XML 的配置元数据展示了这些 bean 的配置是用在顶级 `<beans/>` 元素中的 `<bean/>` 元素完成的。

这些 bean 的定义对应构成应用程序中真实的对象。比如你定义的服务层的对象，数据访问对象（Data Access Object，DAO），表示对象比如 Struts 的 Action 实例，基础设置对象 比如 Hibernate 的 SesstionFactories，JMS 的 Queues 等这些典型的例子。而不用在容 器中定义细粒度的领域模型对象，因为这通常是由 DAO 和业务逻辑负责创建并加载的领域 对象。但是你也可以使用 Spring 和 AspectJ 整合来配置创建在 IoC 容器控制之外的对象。参 考使用在 Spring 中使用 AspectJ 来对领域对象进行依赖注入（8.8.1 节）。

下面的示例展示了基本的基于 XML 的配置元数据的结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xsd">
    <bean id="..." class="...">
        <!-- 这个bean的合作者和配置在这里编写 -->
    </bean>
    <bean id="..." class="...">
        <!-- 这个bean的合作者和配置在这里编写 -->
    </bean>
    <!-- 更多的bean定义在这里编写 -->
</beans>
```

id 属性是一个字符串，用来标识定义的独立的 bean。class 属性定义了 bean 的类型， 需要使用类的完全限定名。id 属性的值指的就是协作对象。指写作对象的 XML 在这个示例 中没有展示；参考依赖（4.4 节）来获取更多信息。

4.2.2 实例化容器

实例化 Spring 的 IoC 容 器 是 很 简 单 的 。 定 位 路 径 或 所 有 路 径 提 供 给 ApplicationContext 的构造方法，实际上是表示资源的字符串，它就允许容器从各种外 部资源比如本地文件系统，Java 的 CLASSPATH 等来加载配置元数据

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml",
    "daos.xml"});
```

> 注意
>
> 在学习过 Spring 的 IoC 容器之后，你可能想更多了解 Spring 的 Resource 抽象，这在第 5 章，资源中会描述，它提供了一个从定义 URI 语法的位置读取输入流的简便机制。特别 是，Resource 路径用来构建应用程序上下文，这会在 5.7 节“应用上下文和资源路径”中 来描述。

下面的示例展示了服务层代码（services.xml）的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xsd">
    <!-- services -->
    <bean id="petStore"
        class="org.springframework.samples.jpetstore.services
        .PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- 这个bean的其它合作者和配置在这里编写-->
    </bean>
    <!-- 更多的service bean的定义在这里编写 -->
</beans>
```

下面的示例展示了数据访问对象的 daos.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xsd">
    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.ibat is.SqlMapAccountDao">
        <!-- 这个bean的其它合作者和配置在这里编写 -->
    </bean>
    <bean id="itemDao"
        class="org.springframework.samples.jpetstore.dao.ibat is.SqlMapItemDao">
        <!-- 这个bean的其它合作者和配置在这里编写 -->
    </bean>
    <!-- 更多数据访问对象的定义在这里编写 -->
</beans>
```

在上面的示例中，服务层代码由类 PetStoreServiceImpl 和两个基于 iBatis（iBatis 现已更名为 MyBatis，译者注）对象/实体映射框架的数据访问对象 SqlMapAccountDao 和 SqlMapItemDao 构成。property 的 name 元素指的是 JavaBean 的属性名，而 ref 元 素指的是其它 bean 定义的名称。id 和 ref 元素之间的这种联系表示了两个协作对象的依赖 关系。要了解更多配置对象依赖的信息，可以参考依赖（4.4 节）。

4.2.2.1 处理基于 XML 的配置元数据

跨越多个 XML 文件中定义 bean 是很有用的。通常每个独立的 XML 配置文件代表了一 个逻辑层或架构中的一个模块。

你可以使用 ApplicationContext 的构造方法从所有的 XML 文件片段中来加载 bean。这个 构造方法可以接收多个 Resource 位置，这在之前的部分都已经看到了。另外，可以使用一个或多个 `<import/>` 元素来从另外的一个或多个文件中加载 bean。比如：

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>
    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

在 这 个示 例 中， 外部 的 bean 通 过 三个 文件 来 加载 ， 分 别 是 services.xml ，messageSource.xml 和 themeSource.xml。所有的位置路径都是相对于该引用文件的， 所以 service.xml 必须在和引用文件 相同路 径中 或 者 是 类 路 径 下 ， 而 messageSource.xml 和 themeSource.xml 必 须 是 在 位 于 引 用 文 件 下 一 级 的 resources 路径下。正如你看到的，前部的斜杠被忽略了，这是由于路径都是相对的，最 好就不用斜线。文件的内容会被引入，包括顶级的 `<beans/>` 元素，根据 Spring 的 Schema 或 DTD，它必须是有效 bean 定义的 XML 文件。

> 注意
>
> 在父目录中使用相对路径“../”来引用文件，这是可能的，但是不推荐这么做。这么做了会创建一个文件，它是当前应用程序之外的一个依赖。 特别是，这种引用对于“classpath：”的 URL（比如，“classpath:../service.xml”）是不推荐的，运行时的解析过 程会选择“最近的”类路径根目录并且会查看它的父目录。类路径配置的修改可能会导致去选择一个不同的，不正确的目录。 你也可以使用资源位置的完全限定名来代替相对路径：比如，“file: C:/config /services.xml” 或“classpath:/config/services.xml”。这样的话，要注意你会耦合应用程序的配置到指定 的绝对路径。对于绝对路径，一般最好是保持一个间接的使用，比如通过占位符“${...}”， 这会基于运行时环境的 JVM 系统属性来解决。

4.2.3 使用容器

ApplicationContext 是能维护不同的 bean 和它们依赖注册的高级工厂接口。使用 `T getBean(Stringname, Class<T> requiredType)` 方法你可以获取 bean 的实例。

ApplicationContext 允许你读取 bean 并且访问它们，如下所示：

```
// 创建并配置bean
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml",
    "daos.xml"});
// 获取配置的实例
PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.class);
// 使用配置的实例
List userList service.getUsernameList();
```

使用 getBean()方法来获取 bean 的实例。ApplicationContext 接口有一些其它方法来获取 bean，但最好应用程序代码不使用它们。事实上，应用程序代码应该没有 getBean()方法的调用，那么就没有对 Spring API 的依赖了。比如，Spring 和 Web 框架整 合时，提供了对各种 Web 框架类库的依赖注入，不如控制器和 JSF 管理的 bean。

4.3 Bean 概述

Spring 的 IoC 容器管理一个或多个 bean。这些 bean 通过提供给容器的配置元数据被创 建出来，比如，在 XML 中的 `<bean/>` 定义的形式。

在容器本身，这些 bean 代表了 BeanDefinition 对象，它们包含（在其它信息中） 下列元数据：

打包的类限定名：就是这些 bean 的真正实现类。

bean 的行为配置元素，这表示了 bean 在容器中（范围，生命周期回调等等）应该是怎 样的行为。

对其它 bean 的引用，这是该 bean 工作所需要的；这些引用通常被称为合作者或依赖。

在新被创建的对象中的其它配置设置，比如，管理连接池的 bean 中使用的连接数或连 接池限制的大小。

元数据翻译成一组属性集合来构成每一个 bean 的定义。

表 4.1 bean 定义

属性	解释章节
class	4.3.2 节，“实例化 bean”
name	4.3.1 节，“命名 bean”
scope	4.5 节，“bean 的范围”
constructor arguments	4.4.1 节，“依赖注入”
properties	4.4.1 节，“依赖注入”
autowiring mode	4.4.5 节，“装配合作者”
lazy-initialization mode	4.4.4 节，“延迟初始化 bean”
initialization method	4.6.1.1 节，“初始化回调”
descruction method	4.6.1.2 节，“销毁回调”

此外，bean 的定义还包含如何创建特定 bean 的信息，ApplicationContext 的实现类允许用户将容器外部创建的已有对象的注册。这可以通过 getBeanFactory()方法访问 ApplicationContext 的 BeanFactory 来完成， 它 会 返 回 BeanFactory 的实现类 DefaultListableBeanFactory 。 DefaultListableBeanFactory 通 过 registerSingleton(..)和 registerBeanDefinition(..)方法来支持这种注册。 而典型的应用程序只和通过元数据定义的 bean 来工作。

4.3.1 命名 bean

每个 bean 有一个或者多个标识符。这些标识符必须在托管 bean 的容器内唯一。通常 一个 bean 只有一个标识符，但如果它需要多个的话，其它的可以被认为是别名。

在基于 XML 的配置元数据中，你可以使用 id 和/或 name 属性来指定 bean 的标识符。 id 属性允许你指定一个准确的 id。按照管理这些名称是字母和数字（‘myBean’，‘fooService’ 等），但是可能是特殊字符。如果你想为 bean 引入别名，你也可以在 name 属性中来指定，

通过逗号（,），分号（;）或空格来分隔。作为对历史的说明，在 Spring 3.1 版之前，id 属性是 xsd:ID 类型，只限定为字符。在 3.1 版本中，现在是 xsd:string 类型了。要注意 bean 的 id 的唯一性还是被容器所强制的，但是对于 XML 处理器却不是。

当然你也可以不给 bean 提供 name 或 id。如果没有提供明确的 name 或 id，那么容器会为这个 bean 生成一个唯一的名称。然而，如果你想通过名称来参考这个 bean，那么可以 使用 ref 元素或者是服务定位器（4.15.2 节）风格的查找，你必须提供一个名称。不提供名称的动机是和使用内部 bean（4.4.2.3 节）或自动装备合作者（4.4.5 节）相关的。

Bean 的命名约定

该约定是用于当命名 bean 时，对实例字段名称的标准 Java 约定。也就是说，bean 的名称以小写字母开始，后面是驼峰形式的。这样的命名可以是（没有引号）‘accountManager’，‘accountService’，‘userDao’，‘loginController’ 等。

一致的命名 bean 可以使你的配置信息易于阅读和理解，如果你使用 Spring 的 AOP， 当应用通知到一组相关名称的 bean 时，它会给你很大的帮助。

4.3.1.1 在 bean 定义外面起别名

在 bean 定义的本身，你可以为 bean 提供多于一个的名称，使用一个由 id 属性或 name 属性中的任意名称数指定的名称组合。这些名称对于同一个 bean 来说都是相等的别名，在 一些情况下，这是很有用的，比如在应用程序中允许每个组件参考一个共同的依赖，可以使 用为组件本身指定的 bean 的名称。

然而，在 bean 定义时指定所有的别名是不够的。有事可能想在任意位置，为一个 bean 引入一个别名。这在大型系统中是很常见的例子，其中的配置信息是分在每个子系统中的， 每个子系统都有它自己的对象定义集合。在基于 XML 的配置元数据中，你可以使用 `<alias/>` 元素来完成这项工作。

```
<alias name="fromName" alias="toName"/>
```

在这个示例中，在相同容器中的名称为 fromName 的 bean，在使用过这个别名定义后， 也可以使用 toName 来指引。

比如，在子系统的配置元数据中，A 可能要通过名称‘subsystemA-dataSource’来指向 数据源。为子系统 B 的配置元数据可能要通过名称‘subsystemB-dataSource’来指向数据源。

当处理使用了这两个子系统的主程序时，主程序要通过名称‘myApp-dataSource’来指向数 据源。那么就需要三个名称指向同一个对象，那么就要按照下面的别名定义来进行添加

MyApp 的配置元数据：

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

现在每个组件和主程序都可以通过一个唯一的保证不冲突的名称来还有其它任意定义（更有效地是创建命名空间）来参照数据源了，当然它们参照的是同一个 bean。

4.3.2 实例化 bean

bean 的定义信息本质上是创建一个或多个对象的方子。当需要一个 bean 时，容器查找 这些方子来查找命名的 bean，并且使用由 bean 定义信息封装的配置元数据来创建（或获得） 一个真实的对象。

如果你使用基于 XML 的配置元数据，你为要被实例化的对象所指定的类型（或类）是 `<bean/>` 元素中 class 属性。这个 class 属性，是 BeanDefinition 实例内部的 cla ss 属性，通常是强制要有的。（对于特例，可以参考 4.3.2.3 节，“使用实例的工厂方法来实例化”和 4.7 节，“Bean 定义的继承”）你可以以两种方法之一来使用 class 属性：

典型的是，指定要被构造的 bean 类，容器本身直接通过反射调用它的构造方法来创建 bean，也就是和 Java 代码中使用 new 操作符是相同的。

指定包含 static 工厂方法的真实的类，会被调用来创建对象，在一些不太常见的情况下，容器会调用类的 static 工厂方法来创建 bean。从调用 static 工厂方法返回的对象类型可能 是和另外一个类完全相同的类。

内部类名称

如果你想为 static 嵌入的类配置 bean，你需要使用内部类的二进制名称。 比如，如果在 com.example 包下有一个 Foo 类,而这个 Foo 类有一个 static 的内部类 Bar，那么在定义 bean 时’class’属性的值就会是...

```
com.example.Foo$Bar
```

请注意名称中$符号的使用，用来从外部类名中分隔内部类名。

4.3.2.1 使用构造方法实例化

当你使用构造方法来创建 bean 时，所有普通的类的使用都和 Spring 兼容。也就是说， 开发中的 bean 不需要实现任何特定的接口或以特定的方式来编码。仅简单指定 bean 的类 就足够了。但基于你使用的是什么类型的 IoC，就可能需要一个默认（空的）构造方法。

Spring 的 IoC 容器可以虚拟地管理任意的你想让它管理的类；而不仅仅限于管理真正的 JavaBean。很多 Spring 用户喜欢在容器中使用有默认（无参数）构造方法和在其后有适当 setter 和 getter 方法的真正 JavaBean。你也可以在容器中使用很多异样的，非 bean 样式的类。比 如，需要使用遗留的连接池，但是它没有符合 JavaBean 的规范，Spring 也能照样管理它。

基于 XML 的配置元数据，你可以如下来定义 bean：

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

关于提供构造方法参数（如果需要）和在对象被构造后，设置对象实例属性机制的详情， 请参考 4.4.1 节依赖注入。

4.3.2.2 使用静态工厂方法来实例化

当使用静态工厂方法来定义 bean 的时候，可以使用 class 属性来指定包含 static 工厂 方法的类，而名为 factory-method 的属性来指定静态方法。你应该调用这个方法（还可 以有可选的参数）并返回一个实际的对象，随后将它视为是通过构造方法创建的一样。在遗留代码中，这样定义 bean 的使用方式之一是调用 static 工厂。

下面 bean 的定义指定了要通过调用工厂方法生成的 bean。这个定义没有指定返回对象 的类型（类），仅仅是包含工厂方法的类。在本例中，createInstance()方法必须是静 态方法。

```
<bean id="clientService" class="examples.ClientService" factory-method="createInstance"/>
public class ClientService {
    private static ClientService clientService = new
        ClientService();
    private ClientService() {}
    public static ClientService createInstance() {
        return clientService;
    }
}
```

关于提供（可选的）参数到工厂方法并在对象由工厂返回后设置对象实例属性的机制详 情，请参考 4.4.2 节深入依赖和配置。

4.3.2.3 使用实例工厂方法来实例化

和使用静态工厂方法（4.3.2.2 节）实例化相似，使用实例工厂方法实例化是要调用容器 中已有 bean 的一个非静态的方法来创建新的 bean。要使用这个机制，请把 class 属性留 空，但在 factory-bean 属性中指定当前（或父/祖先）容器中 bea n 的名字，该 bean 要 包含被调用

来创建对象的实例方法。使用 factory-method 方法来设置工厂方法的名称。

```
<!-- 工厂bean，包含名为createInstance()的方法 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 为locator bean注入任意需要的依赖 -->
</bean>
<!-- 通过工厂bean来创建的bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
public class DefaultServiceLocator {
    private static ClientService clientService = new
        ClientServiceImpl();
    private DefaultServiceLocator() {}
    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

一个工厂类也可以有多于一个工厂方法，比如下面这个：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!--为locator bean注入任意需要的依赖 -->
</bean>
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
<bean id="accountService" factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
public class DefaultServiceLocator {
    private static ClientService clientService = new
    ClientServiceImpl();
    private static AccountService accountService = new
    AccountServiceImpl();
    private DefaultServiceLocator() {}
    public ClientService createClientServiceInstance() {
        return clientService;
    }
    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

这个方法展示了工厂 bean 本身可以通过依赖注入（DI）被管理和配置。请参考 4.4.2 节

深入依赖和配置。 注意在 Spring 文档中，工厂 bean 指的是在 Spring 容器中配置的 bean，可以通过实例（4.3.2.3节）或静态（4.3.2.2 节）工厂方法来创建对象。与此相反的是，FactoryBean（注意大小写）指的是 Spring 特定的 FactoryBean（4.8.3 节）

4.4 依赖

典型的企业级应用程序不是由单独对象（或 Spring 中的 bean）构成的。尽管最简单的 应用程序有一些对象协同工作来表示终端用户所看到的连贯的应用。下一节会解释如何去为 完全实现的应用程序定义一组独立的 bean，其中对象间相互协作来达到目标。

4.4.1 依赖注入

依赖注入（DI）是对象定义它们依赖的过程，也就是说，要和它们协同工作其它对象， 仅仅可以通过构造方法参数，工厂方法参数，或者是在工厂方法返回的对象或被构造好后，

为对象实例设置的属性。容器当创建好 bean，随后就会注入那些依赖。这个过程从根本上 来说是反向的，因此命名为控制反转（IoC），bea n 本身直接使用构造好的类或服务定位器 模式来控制实例或它的依赖的所在位置。

应用了 DI 原则，代码就干净多了，当为对象提供它们依赖的时候，解耦是很有效率的。

对象不再去检查它的依赖，也不需要知道位置或依赖的类。因此，类就很容易被测试，特别是当依赖是接口或抽象基类的时候，这允许在单元测试中使用 stub 或 mock 的实现类。

DI 存在两种主要的方式，基于构造方法的依赖注入（4.4.1.1 节）和基于 setter 方法的依赖注入（4.4.1.2 节）。

4.4.1.1 基于构造方法的依赖注入

基于构造方法的依赖注入是容器调用构造方法和一组参数完成的，每个都表示着一个依 赖。使用特定的参数来调用 static 工厂方法构造 bean 基本也是相同的，这种说法把给构 造方法的参数和给 static 工厂方法的参数相类似。下面的示例展示了一个仅使用构造方 法进行依赖注入的饿类。注意这个类没有什么特殊之处，就是一个 POJO 而且没有对容器特 定接口，基类或注解的依赖。

```
public class SimpleMovieLister {
    // SimpleMovieLister对MovieFinder有依赖
    private MovieFinder movieFinder;
    // 这个构造方法使得Spring容器可以'注入'MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // 业务逻辑就可以'使用'注入的MovieFinder了，代码就省略了...
}
```

构造方法参数解析

构造方法参数解析匹配使用参数的类型。如果在 bean 的构造方法参数不存在潜在的歧义，那么当 bean 被实例化的时候，定义的构造方法参数的顺序就是被提供到适当构造方法 参数的顺序。请看下面的类 ：

```
package x.y;
public class Foo {
    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

没有潜在的歧义存在，假设 Bar 和 Baz 类没有继承关系。因此下面的配置就可以使用 了，而且并不需要明确地在 `<constructor-arg/>` 元素中去指定构造方法参数的索引和/ 或类型。

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>
    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
</beans>
```

当另外一个 bean 被引用时，类型是明确的，那么匹配就能成功（前面示例中也是这样的）。当使用简单类型时，比如 `<value>true<value>`，Spring 不能决定值的类型，所以 没有帮助是不能匹配的。看下面的示例：

```
package examples;
public class ExampleBean {
    //计算最佳答案的年数
    private int years;
    // 生命，宇宙，所有问题的答案
    private String ultimateAnswer;
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

构造方法参数类型匹配

在上面的情形下，如果你使用 type 属性明确地为构造方法参数指定类型的话，容器可 以进行匹配简单的类型。比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

构造方法参数索引

使用 index 属性来指定明确的构造方法参数索引。比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

此外，为了解决多个简单值的歧义，如果构造方法有两个相同类型的参数时，指定所以 可以解决歧义。注意索引是基于 0 开始的。

构造方法参数名称

在 Spring 3.0 中，你也可以使用构造方法参数名称来消除歧义：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

要记住要使得这种方式可用，代码必须和内嵌的调试标识一起编译，那样 Spring 才可 以从构造方法中来查找参数。如果没有和调试标识（或不想）一起编译，那么可以使用 JDK 的注解 @ConstructorProperties 来明确构造方法参数。那么示例代码就如下所示：

```
package examples;
public class ExampleBean {
    // 忽略属性
    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

4.4.1.2 基于 setter 方法的依赖注入

基于 setter 方法的依赖注入由容器在调用过无参数的构造方法或无参数的 static 工厂 方法来实例化 bean 之后，再调用 bean 的 setter 方法来完成的。

下面的示例展示了一个仅仅能使用纯 setter 方法进行依赖注入的类。这是一个常见的 Java 类，是一个没有依赖容器指定的接口，基类或注解的 POJO。

```
public class SimpleMovieLister {
    // SimpleMovieLister对MovieFinder有依赖
    private MovieFinder movieFinder;
    // setter方法可以让Spring容器来'注入'MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // 真正'使用'注入的MovieFinder的业务逻辑代码被省略了...
}
```

ApplicationContext 对它管理的 bea n 支持基于构造方法和 setter 方法的依赖注入。

也 支 持 在 一 些 依 赖 已 经 通 过 构 造 方 法 注 入 之 后 再 进 行 setter 方 法 注 入 。 以

BeanDefinition 形式来配置依赖，使用 PropertyEditor 实例将属性从一种形式格式 化到另一种。但很多 Spring 的用户不直接（编程时）使用这些类，而是使用 XML 文件来定 义，之后会在内部转换这些类的实例，需要加载整个 Spring IoC 容器的实例。

基于构造方法还是 setter 方法进行依赖注入？因为你可以将二者混淆，那么对于基于构造方法或是 setter 方法的依赖注入，有一个很好的规则就是为强制依赖使用构造方法参数，而对于可选参数使用 setter 方法。要注意在 setter 方法上使用注解@Required（4.9.1 节），可用于 setter 方法所需的依赖。

Spring 团队通常主张使用 setter 方法注入，因为大量的构造方法参数会使程序变得非 常笨拙，特别是当属性为可选的时候。Setter 方法会让该类的对象今后适合于重新配置或 重新注入。通过 JMX Mbean（第 23 章）来管理就是一个令人关注的用例。

一些人纯粹赞成构造方法注入。提供所有对象的依赖意味着对象在完全初始化状态 时，通常要返回客户端（调用）代码。这个缺点会使得对象变得不适合去重新配置或重新注入。

使用依赖注入的时候要特别注意一些类。当要选择处理没有源代码时的第三方类库 的时候。遗留的类可能没有暴露任何 setter 方法，而构造方法注入则是唯一可用的依赖 注入方式。

4.4.1.3 解决依赖过程

容器按如下步骤来解决 bean 的依赖：

1. ApplicationContext 和描述了所有 bean 的配置元数据一起被创建并初始化。配 置元数据可以通过 XML，Java 代码或注解来指定。
2. 对于每一个 bean 来说，它的依赖被表述为属性，构造方法参数的形式，如果你使用 了静态工厂方法来代替构造方法，那么还会是静态工厂方法参数的形式。当 bean 被实际创建时，这些依赖被提供给 bean。
3. 每个属性或构造方法参数就是一个要被设置的值或者是容器中其它 bean 的引用。
4. 每个属性或构造方法参数值会被转换特定格式的形式，去匹配属性或构造方法参数的类型。默认情况下，Spring 可以转换给定的字符串格式的值到内建的类型，比如 int，long， String，boolean 等。

当容器被创建时，Spring 容器会来验证每个 bean 的配置，包括验证 bean 的引用属性是 否是一个合法的 bean。然而，bean 属性本身直到 bean 真正被创建出来后才被设置进去。 当容器被创建时，bean 的范围是单例时，会被设置成预实例（默认情况）来创建。范围在 4.5 节，“bean 的范围”部分来解释。否则，bean 就会当被请求的时候被创建。一个 bean 的创建潜在地会引起一系列 bean 被创建，因为 bean 的依赖和它依赖的依赖（等等）要不 创建和定义出来。

循环依赖 如果你使用主要的构造方法注入，就可能会引起不可解决的循环依赖情形。 比如，类 A 需要通过构造方法注入获得类 B 的实例，而类 B 也需要通过构造方法注入获得类 A 的实例。如果把类 A 和类 B 进行相互注入，Spring 的 IoC 容器会在运行时检 测到这是循环引用的情况，并且抛出 BeanCurrentlyInCreationException 异常。

一个可行的方案是编辑一些要配置的类的源码，通过 setter 方法而不是构造方法进 行注入。而且，避免构造方法注入并仅仅使用 setter 方法注入。换句话说，尽管这不是推荐做法，你可以通过 setter 方法注入来配置循环依赖。

不像典型的用例（没有循环依赖），在 bean A 和 bean B 之间的循环依赖强制一个 bean 被注入到另一个中会先于被完全初始化自己（就是经典的鸡和蛋的问题）。

通常情况下你可以信任 Spring 做出正确的事情。它会在容器加载时来检测配置问题， 比如引用了一个不存在的 bean 和循环依赖问题。当 bean 被实际创建出来后，Spring 设置属 性和解决依赖会尽量地晚些。这就意味着 Spring 容器已经加载正确了但晚些时候可能会生 成异常，比如当你请求一个创建时发生问题的对象或者是它的依赖发送问题。例如，bean 因为丢失或非法属性而抛出异常。这种潜在的一些配置问题的可见度延迟也就是为什么 ApplicationContext 的实现类默认情况都是预实例的单例 bean。在前期的时间和内存 消耗中，在 bean 真正需要前来创建这些 bean，当 ApplicationContext 被创建的时候， 你会发现这些配置问题，这还不晚。你也可以覆盖这个默认的行为，那么单例 bean 就会延 迟加载，而不是预实例的了。

如果没有循环依赖的存在，当一个或多个协作的 bean 被注入到一个独立的 bean 时， 每个协作的 bean 就会在被注入之前完全被配置好。这就意味着如果 bean A 对 bean B 有依 赖，那么 Spring 的 IoC 容器会完全配置 bean B 而优先于调用 bean A 中的 setter 方法。换句 话说，bean 被实例化了（而不是预实例的单例 bean），它的依赖才被注入，相关的生命周 期方法（比如配置初始化方法（4.6.1.1 节）或 InitializingBea n 的回调方法（4.6.1.1 节））才被调用。

4.4.1.4 依赖注入示例

下面的示例使用了基于 XML 的配置元数据来进行基于 setter 方法的依赖注入。Spring XML 配置文件的一小部分来指定几个 bean 的定义：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- 使用嵌入<ref/>的元素来进行setter方法注入<ref/> -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>
    <!-- 使用整洁的'ref'属性来进行setter方法注入-->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    private AnotherBean beanOne; private YetAnotherBean beanTwo; private int i;
    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }
    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

上面的示例中，setter 方法被声明为匹配 XML 文件中指定的属性。下面的示例使用基于 构造方法的依赖注入：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- 使用嵌入的<ref/>元素进行构造方法注入 -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>
    <!-- 使用整洁的'ref'属性来进行构造方法注入 -->
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg type="int" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    private AnotherBean beanOne; private YetAnotherBean beanTwo; private int i;
    public ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

在 bean 中指定的构造方法参数会被用于 ExampleBean 的构造方法参数。 现在考虑一下这个示例的变化情况，要代替使用构造方法，Spring 被告知调用 static

工厂方法并返回一个对象的实例：

```
    <bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    // 私有的构造方法
    private ExampleBean(...) {
        ...
    }
    // 静态工厂方法; 这个方法的参数可以被认为是要返回bean的依赖,
    // 而不管那些参数是如何被使用的。
    public static ExampleBean createInstance (AnotherBean anotherBean,
        YetAnotherBean yetAnotherBean, int i) {
        ExampleBean eb = new ExampleBean (...);
        // 其它的操作...
        return eb;
    }
}
```

static 工厂方法的参数通过 `<constructor-arg/>` 元素来提供，这和构造方法已经 被实际调用是完全一致的。由工厂方法返回的类的类型不需要和包含 static 工厂方法类 的类型相同，尽管在本例中是相同的。实例（非静态）工厂方法可以被用于本质上相同的方 式（除了 factory-bean 属性的使用，来代替 class 属性），所以这里不讨论它们的细节。

4.4.2 深入依赖和配置

正如之前章节中所提到的，你可以定义 bean 的属性和构造方法参数作为其它被管理 bean（协作者）的引用，或者作为内联值的定义。出于这个目的，Spring 的基于 XML 的配置元数据支持使用 `<property/>` 和 `<constructor-arg/>` 元素的子元素类型。

4.4.2.1 直接值（原生类型，String，等）

元素的 value 属性指定了属性或构造方法参数，这是人们可以阅读的 字符串的表达。正如之前提到过的（6.4.2 节），JavaBean 的 PropertyEditor 可以用来转 换这些字符串值从 String 到属性或参数的真实类型。

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="c
    <!-- setDriverClassName(String)方法调用的结果 -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

对于更简洁的 XML 配置，下面的示例使用了 p-命名空间（4.4.2.6 节）。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xs d">
    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-metho
</beans>
```

上面的 XML 非常简洁；然而，错误之处会在运行时被发现而不是设计的时候，除非在 你创建 bean 的时候，使用如 IntelliJ IDEA 或 SpringSource Tool Suite（STS，SpringSource 组织开发的工具套件）这样的 IDE 来支持自动地属性补全。这样的 IDE 帮助是强烈建议使用的。

你也可以这样来配置 java.util.Properties 实例：

```
<bean id="mappings" class="org.springframework.beans.factory.config.PropertyPlac
    eholderConfigurer">
    <!-- 作为java.util.Properties类型 -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver jdbc.url=jdbc:mysql://localhost:3
        </value>
    </property>
</bean>
```

Spring 容器会使用 JavaBean 的 PropertyEditor 机制来转换 `<value/>` 元素中的文本 到 java.util.Properties 实例。这是一个很好的捷径，也是 Spring 团队少有喜欢使用 嵌套的 `<value/>` 元素而不是 value 属性方式的地方之一。

idref 元素

idref 元素是一种简单防错的形式来传递容器中另外一个 bean 的 id（字符串值而不是引用）到 `<constructor-arg/>` 或 `<property/>` 元素中。

```
<bean id="theTargetBean" class="..."/>
<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

上面定义 bean 的代码片段是和下面的片段完全等价（在运行时）：

```
<bean id="theTargetBean" class="..." />
<bean id="client" class="...">
    <property name="targetName" value="theTargetBean" />
</bean>
```

第一种形式比第二种形式可取，因为使用 idref 标签允许容器在部署时来验证被引用 的，命名的 bean 是否真的存在。第二种形式下，没有对传递给 client bea n 的 targetName 属性执行验证。错误仅仅是在 client bean 被真正实例化的时候才会被发现（和很多可能 致命的结果）。如果 client bean 是 prototype（4.5 节）类型的 bean，这个错误和结果异常 可能仅仅在容器被部署很长一段时间后才会被发现。

此外，如果被引用的 bean 在同一个 XML 单元中，bean 的名称就是 bean 的 id，那么你 还可以使用 local 属性，这允许 XML 解析器本身在 XML 文档解析的时候，及早地来验证 bean 的 id。

```
<property name="targetName">
    <!-- id为'theTargetBean'的bean必须存在;否则就会抛出异常 -->
    <idref local="theTargetBean"/>
</property>
```

`<idref/>` 元素带来的一个相同之处（最少是 Spring 2.0 版本以前的）是值在 ProxyFactoryBean 定义的 AOP 拦截器的配置中。当你指定拦截器名称来防止拼错拦截 器的 id 时，可以使用 `<idref/>` 元素。

4.4.2.2 引用其它 bean（协作者）

ref 元素是 `<constructor-arg/>` 或 `<property/>` 定义元素中最后的一个。在这里 你可以为 bean 设置指定属性的值，来引用被容器管理的另外一个 bean（协作者）。被引用 的 bean 就是要设置属性的这个 bean 的一个依赖，并且是初始化的作为属性设置之前的点 播。（如果协作者是一个单例的 bean，它可能已经被容器初始化过了。）所有引用最终都会 被引用到一个对象中。范围和验证基于通过 bean，local 或 parent 属性指定 id/name 的其它对象。

通过 `<ref/>` 标签的 bean 属性来指定目标 bean 是最常用的方式，并允许创建引用到相 同容器或父容器的任意 bean 中，而不管它们是不是在同一个 XML 文件中。bean 属性的值 可能和目标 bean 的 id 属性相同，或者是目标 bean 的 name 属性值之一。

```
<ref bean="someBean"/>
```

通过 local 属性来指定目标 bean 是利用了 XML 解析器的能力，来验证 XML 相同文件 内的 id 引用。local 属性的值必须和目标 bean 的 id 属性一致。如果没有在相同的文件内 发现匹配的元素，那么 XML 解析器会报告问题。因此，如果目标 bean 在同一个 XML 文件 中的话，使用 local 这种形式是最佳选择（为了更早地知道错误）。

```
<ref local="someBean"/>
```

通过 parent 属性来指定目标 bea n 会为 bea n 创建引用，它是在当前容器的父容器中的。parent 属性的值可能和目标 bean 的 id 属性相同，或者是目标 bean 的 name 属性值 之一，而且目标 bean 必须在当前容器的父容器中。当有一个容器继承关系，可以使用这个 bean 的引用，或在你想使用和父 bean 有相同名称的代理包装一个父容器中已有 bean 时。

```
<!-- 在父上下文中-->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- 在这里插入所需要的依赖 -->
</bean>
<!-- 在子（后继）上下文中 -->
<bean id="accountService" <-- bean名称和父bean相同 -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>
        <!-- 注意我们如何引用父bean -->
    </property>
    <!-- 在这里插入其它配置和所需的依赖 -->
</bean>
```

4.4.2.3 内部 bean

在 `<property/>` 或 `<constructor-arg/>` 元素内部的 `<bean/>` 元素的定义被称为内部 bean。

```
<bean id="outer" class="...">
    <!-- 为了替代为目标bean使用引用，简单内联定义目标bean -->
    <property name="target">
    <bean class="com.example.Person"> <!-- 这就是内部bean -->
        <property name="name" value="Fiona Apple"/>
        <property name="age" value="25"/>
    </bean>
    </property>
</bean>
```

内部 bean 的定义不需要定义 id 或 name；容器忽略这些值。也会忽略 scope 标识。内 部 bean 通常是匿名的，而且范围通常是 prototype（4.5.2 节）的。注入内部 bean 到协作 bean 是不可能的，只能到包围它的 bean 中。

4.4.2.4 集合

在 `<list/>`， `<set/>`， `<map/>` 和 `<props/>` 元素中，你可以分别设置 Java 的 Collection 类型 List，Set，Map 和 Properties 的属性和参数。

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop
            key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

Map 类型的 key 或 value 的值，或 set 的值，也可以是下列元素之一：

- bean
- ref
- idref
- list
- set
- map
- props
- value
- null

集合合并

在 Spring 2.0 中，容器支持集合的合并。应用开发人员可以定义父样式的 `<list/>`， `<map/>`， `<set/>` 或 `<props/>` 元素，子样式的 `<list/>`， `<map/>`， `<set/>` 或 `<props/>` 元素继承或重写来自父集合的值。也就是说，子集合的值是合并元素父与子集合中的元素，和子集合元素覆盖父集合中指定值的结果。

本节关于导论父子 bean 合并的机制。如果读者不熟悉父和子 bean 的定义，可能要先去阅读一下相关章节（4.7 节）。

下面的示例说明了集合合并：

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop
                key="administrator">administrator@example.com</prop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- 合并在*子*集合定义中来指定 definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

注意在 child bean 中的 adminEmails 属性下的 `<props/>` 元素中的 merge=true 属性的使用。当 child bean 被容器处理并实例化时，结果实例中有一个 adminEmails Properties 的 集 合 ， 包 含 了 合 并 子 bean 的 adminEmails 集合和父 bean 的 adminEmails 集合。

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

子 bean 的 Properties 集合的值设置继承了从父 `<props/>` 元素而来的所有属性，子 bean 中 support 的值覆盖了父 bean 中的值。

这种合并行为的应用和 `<list/>`， `<map/>` 和 `<set/>` 集合类型很相似。在 `<list/>` 元素特 定的情形下，和 List 集合类型相关的语义，也就是说，ordered 集合值的概念，是要维 护的；父值优先于所有子 list 的值。在 Map，Set 和 Properties 集合类型的情况下，没有顺序的存在。因此对于容器内部使用的，和 Map，Set 和 Properties 实现类型相关的 集合类型没有排序语义的作用。

集合合并的限制

不能合并不同类型的集合（比如 Map 和 List），如果你要尝试这么去做，那么就会抛 出 Exception。merge 属性必须在低级的，继承的，子 bean 中来指定；在父集合中指定 merge 属性是冗余的，也不会看到想要的合并结果。合并特性仅在 Spring 2.0 和更高版本中 可用。

强类型集合（Java 5 以上版本）

在 Java 5 或更高版本中，你可以使用强类型集合（使用泛型）。也就是说，可以声明一 个 Collection 类型，它可以仅仅包含 String 元素（作为示例）。如果你使用 Spring 来对强类型的 Collection 依赖注入到 bean 中，你可以利用 Spring 的类型转换来支持这样 强类型 Collection 实例的元素，在被加到 Collection 之前，可以转换成合适的类型。

```
public class Foo {
    private Map<String, Float> accounts;
    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

当 foo bean 的 accounts 属性准备好注入时，关于强类型元素 `Map<String, Float>` 类型的泛型信息就通过反射机制准备好了。因此 Spring 的类型转换工具识别到各种元素的 值作为 Float 类型，字符串值 9.99，2.75 和 3.99 被转换成实际的 Float 类型。

4.4.2.5 null 和空字符串

Spring 将属性的空参数当作空 String。下面基于 XML 的配置元数据片段设置了电子邮 件属性为空 String 值（""）。

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

上面的例子和下面的 Java 代码是一样的：exampleBean.setEmail("")。 `<null/>` 元素控制 null 值。比如：

```
<bean class="ExampleBean">
    <property name="email"><null/></property>
</bean>
```

上面的配置和下面的 Java 代码一致：exampleBean.setEmail(null)。

4.4.2.6 使用 p-命名空间的 XML 快捷方式

p-命名空间可以使你使用 bea n 的元素属性，而不是内嵌的 `<property/>` 元素，来描述属 性的值和/或协作的 bean。

Spring 2.0 和后续版本支持使用命名空间（附录 C）的可扩展的配置格式，这是基于 XML的 Schema 定义的。本章中讨论的 bea n 的配置格式是定义在 XML 的 Schema 下的。然而，p-命名空间则不是定义在 XSD 文件下的，仅仅存在于 Spring 的核心中。 下面的示例展示了两个 XML 片段，解析得到相同的结果：第一个使用了标准的 XML 格式，第二个使用了 p-命名空间。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xsd">
    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value=["foo@bar.co](mailto:foo@bar.com)m"/>
    </bean>
    <bean name="p-namespace" class="com.example.ExampleBean" p:email=["foo@bar.co](mailto
</beans>
```

示例展示了 p-命名空间下的属性，在 bean 的定义中称为 email。这就告诉 Spring 来包 含属性声明。正如前面提到的，p-命名空间没有 schema 定义，所以你可以设置属性的名称 和 bean 中属性的名称一样。

下面的示例包含了两个 bean 的定义，两者都有对其它 bean 的引用：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation"http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring -beans-3.0.xsd">
    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>
    <bean name="john-modern" class="com.example.Person" p:name="John Doe" p:spouse-ref="j
    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

正如你所看到的一样，这个例子包含着不仅仅使用了 p-命名空间的属性值，也使用了 特殊格式来声明属性的引用。在第一个 bean 中 使用了 `<property name="spouse" ref="jane"/>` 来创建 john bean 对 jane bean 的引用关系，第二个 bean 中，使用了 p:spouse-ref="jane"作为属性来做相同的事情。本例中，spouse 是属性名，而-ref 部分表明了这不是一个直接值而是一个对其它 bean 的引用。

> 注意
>
> P-命名空间没有标准 XML 格式那么灵活。比如，声明属性引用的格式和以 Ref 结尾的属性相冲突，而标准的 XML 格式则不会。我们建议谨慎选择所用的方法并和开发团队成员 充分地交流，避免产生同时使用这三种方法的 XML 文档。

4.4.2.7 使用 c-命名空间的 XML 快捷方式

和 4.4.2.6 节，“使用 p-命名空间的 XML 快捷方式”相似，c-命名空间是在 Spring 3.1 中 被引入的，允许使用内联属性来配置构造方法参数而不是使用 constructor-arg 元素。

我们来看 4.4.1.1 节，“基于构造方法的依赖注入”中的示例，现在使用 c 命名空间 ：

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
    <-- '传统的'声明方式 -->
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
        <constructor-arg value=["foo@bar.co](mailto:foo@bar.com)m"/>
    </bean>
    <-- 'c-命名空间'声明方式 -->
    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email=["foo@bar.co](
</beans>
```

c:命名空间使用了和 p:（以-ref 结尾的 bean 引用）相同的转换机制通过它们的名称 来设置构造方法参数。这样一来，即便没有在 XSD schema（但是存在于 Spring 核心的内部） 中定义， 它还是需要声明出来。

在极少数的情况下，构造方法参数名称是不可用的（通常如果字节码在编译时没有调试 信息）， 我们可以使用参数索引 :

```
<-- 'c-命名空间'索引声明 -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz">
```

> 注意
>
> 因为 XML 的语法，索引符号需要在其头部使用_作为 XML 属性名称，因为 XML 中属性是不能以数字开头的（尽管一些 IDE 允许这样做）。

在实际运用中，构造方法解析机制（4.4.1.1 节）在匹配参数时是非常有效率的，所以除 非真有需要，我们建议在配置中使用名称符号。

4.4.2.8 复合属性名称

在设置 bean 的属性时，只要路径中的所有组件，除了最后一个属性名称是非 null 的， 可以使用复合或者嵌套的属性名称。参考下面的 bean 定义。

```
<bean id="foo" class="foo.Bar">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

foo bean 有一个 fred 属性，它还有一个 bob 属性，而它仍有一个 sammy 属性，而最 终的 sammy 属性被设置成数值 123。为了让这样的配置可用，foo 的属性 fred，fred 的 属性 bob 在 bean 被构造好后必须不能为 null，否则会抛出 NullPointerException 异常。

4.4.3 使用 depends-on

如果一个 bean 是另外一个 bean 的依赖，通常表明了它会被设置成另外一个的属性。 典型的情况是在基于 XML 的配置元数据中使用 `<ref/>` （4.4.2.2 节）元素来完成。然而，有 时在两个 bean 之间的依赖并不是直接的；比如，类中的静态初始化器需要触发，就像数据 库驱动程序的注册。depends-on 属性可以明确地强制一个或多个 bean 在使用这个元素的 bean 被初始化之前被初始化。下面的示例使用了 depends-on 属性来表示一个独立 bean 的依赖：

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

为了表示对多个 bean 的依赖，提供一个 bean 名称的列表作为 depends-on 属性的值 即可，要用逗号，空白或分号分隔开，它们都是有效的分隔符：

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>
<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

> 注意
>
> 在 bean 定义中的 depends-on 属性可以指定初始化时的依赖，也可以是仅仅是单例（4.5.1 节）bean，对应销毁时的依赖。和给定 bean 定义了 depends-on 关系的依赖 bean 首先被销毁，先于给定的 bean 本身。因此 depends-on 也能控制关闭顺序。

4.4.4 延迟初始化 bean

默认情况下，ApplicationContext 的实现类积极地创建并配置所有单例的 bean （4.5.1 节），作为初始化过程的一部分。通常来说，这种预实例化是非常可取的，因为在配 置或周边环境中的错误可以直接被发现，而不是几小时或几天后去发现。当这种行为不可用时，你可以阻止单例 bean 的预实例化，在 bean 定义中使用延迟初始化来标记一下就可以 了。延迟初始化 bean 告诉 IoC 容器在该 bean 第一次被请求时再来实例化，而不是在启动时 实例化。

在 XML 中，这个行为可以通过 `<bean/>` 元素的 lazy-init 属性来控制；比如：

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

当前者配置被 ApplicationContext 处理时，命名为 lazy 的 bean 并不会在

ApplicationContext 启动时被预实例化，而 not.lazy bean 会被先预实例化。 而到延迟初始化 bean 是一个单例 bean 的依赖时，且这个单例 bean 不是延迟初始化的，那么 ApplicationContext 也会在启动时来创建延迟初始化的 bean，因为它必须满足单 例 bean 的依赖。延迟初始化的 bean 被注入到单例 bean 中的时候它就不是延迟初始化的了。

你也可以在容器级别来控制延 迟初始化，在 `<beans/>` 元素上使用default-lazy-init 属性；比如：

```
<beans default-lazy-init="true">
    <!-- 那么就没有bean是预实例化的了... -->
</beans>
```

4.4.5 自动装配协作者

Spring 容器可以自动装配协作 bean 之间的关系。你可以允许 Spring 自动为你的 bean 来 解析协作者（其它 bean），通过检查 ApplicationContext 的内容即可。自动装配有下 列优势：

自动装配可以显著减少指定属性或构造方法参数的需要。（如 bean 模板的机制，这将在 本章的 4.7 节来讨论，在这方面是有价值的。）

自动装配可以更新配置对象的演变。比如，如果你需要给一个类添加依赖，那个依赖可 以自动被填入而不需要修改配置。因此，自动装配在开发期间是非常有用的，当代码库 变得更稳定时切换到明确的装配也没有否定的选择。

当使用基于 XML 的配置元数据（4.4.1 节）时，你可以为 bean 的定义指定自动装配模 式，在 `<bean/>` 元素上设置 autowire 属性即可。自动装配功能有五种模式。你可以为每个 bean 来指定自动装配，因此可以选择为哪一个来指定自动装配。

表 4.2 自动装配模式

模式	解释
no	（默认情况）没有自动装配。Bean 的引用必须通过 ref 元素来定义。对于大型的部署，修改默认设置是不推荐的，因为明确地指定协作者会给予更多的控制和清晰。某种程度上来说，它勾勒出了系统的结构。
byName	通过属性名称来自动装配。Spring 以相同名称来查找需要被自动装配的 bean。比如，如果 bean 被设置成由名称来自动装配，并含有一个 master 属性（也 就说，有 setMaster(..)方法），Spring 会查找名为 master 的 bean 定义，并且用它来设置属性。
byType	如果 bean 的属性类型在容器中存在的话，就允许属性被自动装配。如果存在多于一个，就会抛出致命的异常，这就说明了对那个 bean 不能使用 byType 进行自动装配。如果没有匹配的 bean 存在，就不会有任何效果；属性不会 被设置。
constructor	和 byType 类似，但是是应用于构造方法参数的。如果在容器中没有确定的构造方法参数类型的 bean 的存在，就会发生致命的错误。

使用 byType 或 constructor 自动装配模式，你可以装配数组和集合类型。这种情况下所有在容器内匹配期望类型的自动装配候选者会被用来满足依赖。如果期望的键类型是String，你可以自动装配强类型的 Map。自动装配的 Map 值会包括匹配期望类型的实例， 而且 Map 的键会包含对应 bean 的名称。

你可以联合自动装配行为和依赖检查，这会在自动装配完成之后执行。

4.4.5.1 自动装配的限制和缺点

当在项目中一直使用时，自动装配是非常不错的。如果自动装配通常是不使用的，它就 可能会迷惑开发人员来使用它去装配仅仅一两个 bean。

考虑一下自动装配的限制和缺点：

- 在 property 和 constructor-arg 中设置的明确依赖通常覆盖自动装配。不能自动 装配所谓的简单属性，比如原生类型，String 和 Class（还有这样简单属性的数字）。这 是由于设计的限制。
- 自动装配没有明确装配那么精确，尽管，在上面的表格中已经说明了，Spring 很小心地避免在有歧义的情况下去猜测，但也可能会有意想不到的结果，在 Spring 管理的对象 中间的关系可能就不会再清晰地说明了。
- 装配信息可能对从 Spring 容器中来生成文档的工具来说不可用。
- 在容器中的多个 bean 定义可能通过 setter 方法或构造方法参数匹配指定的类型进行自动装配。对于数组，集合或 Map，这不一定是一个问题。而对期望简单值的依赖，这 种歧义不能随意解决。如果没有唯一的 bean 可用，就会抛出异常。 在后面一种情况中，你有几种选择：
- 放弃自动装配而使用明确的装配。
- 避免设置它的 autowire-candidate 属性为 false 来自动装配 bean，这会在下一 节中来解释。
- 设置 `<bean/>` 元素的 primary 属性为 true 来指定单独的 bean 作为主要的候选者。
- 如果你使用 Java 5 或更高版本，使用基于注解的配置实现更细粒度的控制，这会在 4.9 节，“基于注解的容器配置”中来解释。

4.4.5.2 从自动装配中排除 bean

在每个 bean 的基础上，你可以从自动装配中来排除 bean。在 Spring 的 XML 格式配置 中，设置 `<bean/>` 元素的 autowire-candidate 属性为 false；容器会把指定的 bean 对自动装配不可用（包含注解风格的配置，比如@Autowired（4.9.2 节））。

你也可以基于 bean 名称的模式匹配来限制自动装配候选者。顶级的 `<beans/>` 元素中的 default-autowire-candidates 属性接受一个或多个模式。比如，为了限制自动装 配候选者到任意名称以 Repository 结尾 bean 的状态，提供*Repository 值。要提供多个模式，把它们定义在以逗号分隔的列表中。Bean 定义中 autowire-candidate 属性的 true 或 false 明确的值通常是优先的，而且对于这些 bean 来说，模式匹配规则是不适用的。 这些技术对那些永远不想通过自动装配被注入到其它 bea n 中的 bea n 来说是很有用的。

这并不意味着未包含的 bean 不能使用自动装配来配置。相反，bean 本身不是自动装配其它 bean 的候选者。

4.4.6 方法注入

在很多应用场景中，很多容器中的 bean 是单例（4.5.1 节）的。当一个单例的 bean 需 要和其它单例的 bean 协作时，或者一个非单例的 bean 需要和其它非单例的 bean 协作时， 典型地做法是通过定义一个 bean 作为另外一个的属性来控制依赖。当 bean 的生命周期不 同时问题就产生了。假设单例 bean A 需要使用非单例（prototype，原型）bean B，或许在 A 的每个方法调用上。容器仅仅创建单例 bean A 一次，因此仅仅有一次机会去设置属性。容 器不能每次为 bean A 提供所需的 bean B 新的实例。

解决方法就是放弃一些控制反转。你可通过实现 ApplicationContextAware 接口 以让 bean A 去感知容器（4.6.2 节）， 而且在每次 bean A 需要 bean B 的实例时，可以让容器 调用 getBean(“B”)（4.2.3 节）去获取（一个新的）bean B。下面的代码就是这种方式的例子：

```
// 使用了有状态的命令风格的类来执行一些处理
package fiona.apple;
// 导入Spring的API
import org.springframework.beans.BeansException;
import org.springframework.context.Applicationcontext;
import org.springframework.context.ApplicationContextAware;
public class CommandManager implements ApplicationContextAware {
    private ApplicationContext applicationContext;
    public Object process(Map commandState) {
        // 抓取相应命令的新实例
        Command command = createCommand();
        // 在命令实例上（希望标记新的）设置状态
        command.setState(commandState);
        return command.execute();
    }
    protected Command createCommand() {
        // 注意Spring API的依赖!
        return this.applicationContext.getBean("command", Command.class);
    }
    public void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

前面的做法是不可取的，因为这些业务代码感知并耦合到了 Spring 框架中。方法注入， 这种 Spring IoC 容器中的有点先进的功能，允许以一种干净的方式来处理这个用例。

你可以在博客文章中来阅读更多关于方法注入的动机。

4.4.6.1 查找方法注入

查找方法注入是容器在其管理的 bean 上来覆盖方法的能力，来返回容器中其它命名 bean 的查找结果。查找，典型的情况是涉及原型 bean，这是在之前章节中描述的情景。Spring Framework 从 CGLIB 类库中，通过使用字节码生成机制实现了这种方法注入，来动态地生成覆盖方法的子类。

> 注意
>
> 要让这些动态子类起作用，在类路径下必须有 CGLIB 的 jar 文件。这些 Spring 容器中的子类不能是 final 类型的，要被覆盖的方法也不能是 final 类型的。而且，测试有 abstract 方法的类要求你自己去编写类的子类并提供 abstract 方法的实现。最后，是 方法注入目标的对象还不能被序列化。

看一下之前代码片段中的 CommandManager 类，你会看到 Spring 容器会动态地覆盖 createCommand()方法的实现。CommandManager 类不会有任何 Spring 的依赖，在重新设计 的例子中你会看到：

```
package fiona.apple;
// 没有Spring API的导入!
public abstract class CommandManager {
    public Object process(Object commandState) {
        // 抓取一个心得合适的Command接口的实例
        Command command = createCommand();
        //在命令实例上（希望标记新的）设置状态
        command.setState(commandState);
        return command.execute();
    }
    // 好的... 但是这个方法的实现在哪儿?
    protected abstract Command createCommand();
}
```

在客户端类中包含要被注入（本例中的 CommandManager）的方法，要被注入的方法 需要从下面的示例中编写一个签名：

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

如果方法是 abstract 的，动态生成的子类就实现这个方法。否则，动态生成的子类覆盖 定义在源类中的确定方法。比如：

```
<!-- 部署为原型的（非单例的）有状态的bean -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- 注入所需要的依赖 -->
</bean>
<!-- commandProcessor使用statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="command"/>
</bean>
```

当需要 command bean 的新的实例时，标识为 commandManager 的 bean 调用它自己的 方法 createCommand()。部署 command bean 为原型的可必须要小心，那确实就是需要 的才行。如果被部署为单例的（4.5.1 节），那么每次会返回相同的 command bean 的实例。

> 提示
>
> 感兴趣的读者可能会发现要使用 ServiceLocatorFactoryBean （在 org.springframework.beans.factory.config 包下）。在 ServiceLocatorFactoryBean 中的使用的方法和其它工具类是相似的，ObjectFactoryCreatingFactoryBean，但是它允许你去指定你自己的查找接口而不是 Spring 特定的查找接口。对这些类查询一下 JavaDoc 文档，还有博客文章来获取 ServiceLocatorFactoryBean 的额外的信息。

4.4.6.2 任意方法的替代

在方法注入中，用处不如查找方法注入大的一种形式，就是使用另外一个方法实现来替 换被容器管理的 bean 中的任意方法的能力。用户可以安全地跳过本节的其它部分，直到你 真正需要这些功能的时候来回过头来看。

使用基于 XML 的配置元数据，对于部署的 bean，你可以使用 replaced-method 元 素来使用另外一个方法替换已有的方法实现。看看下面这个类，有一个我们想要去覆盖的 computeValue 方法：

```
public class MyValueCalculator {
    public String computeValue(String input) {
        // 真正的代码...
    }
    // 其它方法...
}
```

实现了 org.springframework.beans.factory.support.MethodReplacer 接口的类提供新的方法定义。

```
/** 也就是要在MyValueCalculator中实现用来覆盖已有的方法
computeValue(String)
*/
public class ReplacementComputeValue implements MethodReplacer {
    public Object reimplement(Object o, Method m, Object[] args)
    throws Throwable {
        // 获取输入值，并处理它，返回一个计算的结果
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

部署源类的 bean 定义和指定的方法覆盖可以是这样的：

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
<!-- 任意方法替换 -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>
<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

你可以在 `<replaced-method/>` 元素中使用一个或多个包含的 `<arg-type/>` 元素来 指示被覆盖方法的方法签名。如果方法是被重载的并在类中有很多种形式的存在，那么参数 的签名是必须的。为了简便，字符串类型的参数可能是类型全名的一个子串。比如，下面所 有的都匹配 java.lang.String：

```
java.lang.String
String
Str
```

因为用参数的数量用来区分每一个可能的选择通常是足够的，这种快捷方式可以节省大 量的输入，允许你去输入匹配参数类型的最短的字符串。

4.5 Bean 的范围

当创建 bean 时，也就创建了对通过 bean 定义创建的类真正实例的配方。bean 定义的 配方这个概念是很重要的，因为它就意味着，你可以通过配方来创建一个类的多个对象实例。

你不仅可以控制从特定 bean 定义创建出的对象的各个依赖和要配置到对象中的值，也 可以控制对象的范围。这个方法非常强大并且很灵活，你可以选择通过配置创建的对象的范 围，而不必去处理 Java 类级别对象的范围。Bean 可以被定义部署成一种范围：开箱，Spring

Framework 支持五种范围，里面的三种仅仅通过感知 web 的 ApplicationContext 可用。

下列的范围支持开箱。你也可以创建自定义范围（4.5.5 节）。

表 4.3 Bean 的范围

范围	描述
单例（4.5.1 节）	（默认的）为每一个 Spring 的 IoC 容器定义一个 bean 为独立对象的实例。
原型（4.5.2 节）	定义可以有任意个对象实例的 bean。
请求（4.5.4.2 节）	定义生命周期为一个独立 HTTP 请求的 bean；也就是说，每一个 HTTP 请求有一个 bean 的独立的实例而不是一个独立的 bean。仅仅在可感知 Web 的 Spring ApplicationContext 中可用。
会话（4.5.4.3 节）	定义一个生命周期为一个 HTTP Session 的独立的 bean。仅仅在可感知 Web 的 Spring ApplicationContext 中可用。
全局会话（4.5.4.4 节）	定义一个生命周期为一个全局的 HTTP Session 的独立的 bean。典型地是仅仅 使用 portlet 上 下文 时可 用。 仅仅 在可 感知 Web 的 Spring ApplicationContext 中可用。

线程范围的 bean

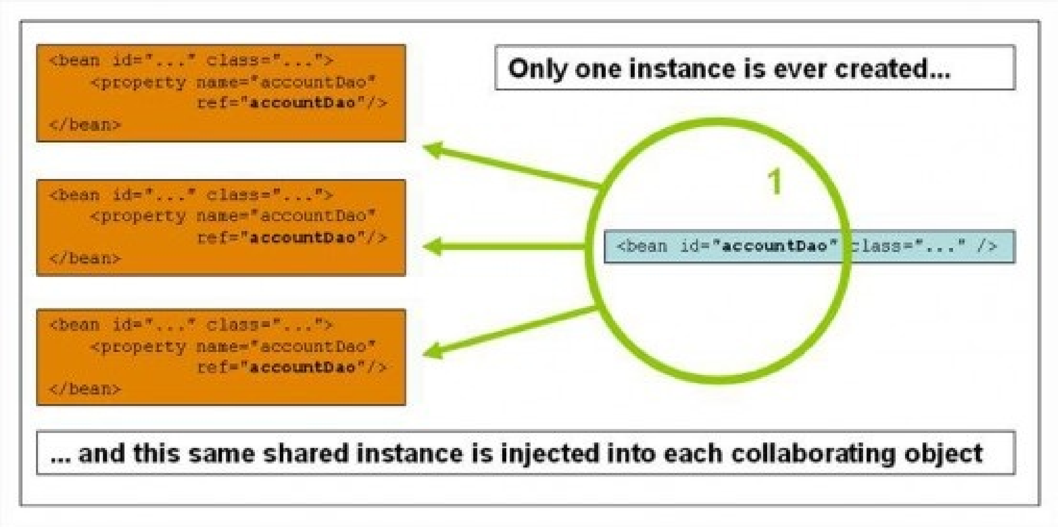
在 Spring 3.0 当中，线程范围是可用的，但是它默认是不注册的。要获取更多信息，请 参考 SimpleThreadSc ope 的 JavaDoc 文档。对于如何注册这个范围或其它自定义范围的做法，

请参考 4.5.5.2 节，“使用自定义范围”。

4.5.1 单例范围

仅仅共享被管理的 bean 的一个实例，所有使用一个或多个 id 来匹配 bean 的结果对 bean 请求，由 Spring 容器返回指定 bean 的实例。

另外一种方式，当你定义一个范围是单例的 bean 后，Spring 的 IoC 容器通过 bean 的定 义创建了这个对象的一个实例。这个独立的实例存储在单例 bean 的缓存中，所有对这个命 名 bean 的后续的请求和引用都返回缓存的对象。



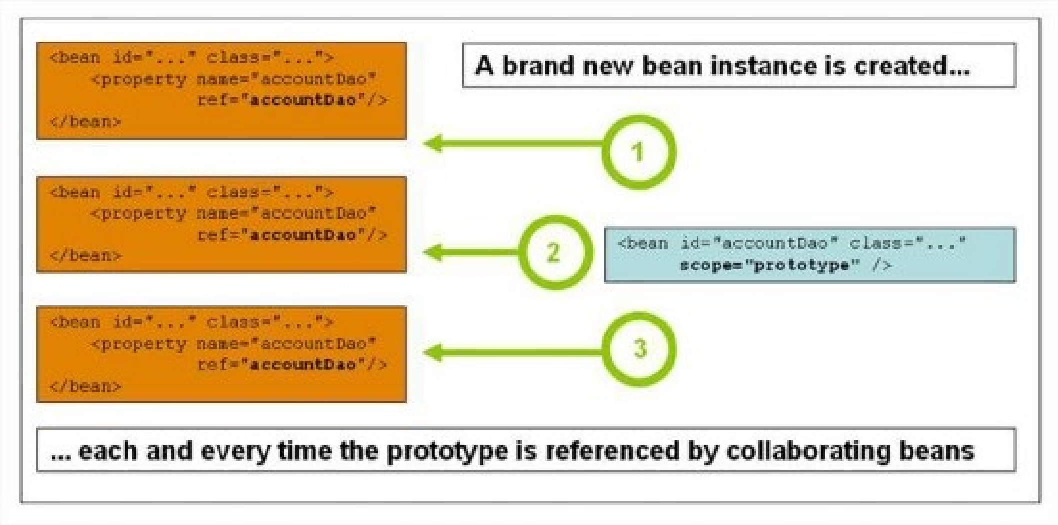
Spring 中对单例 bean 的概念和四人帮（Gang of Four，GoF）设计模式书中定义的单例 模式是不同的。GoF 的单例硬编码了对象的范围，也就是特定的类仅仅能有一个实例被每一 个 ClassLoader 来创建。Spring 中单例 bean 的范围，是对每一个容器和 bean 来说的。这 就意味着在独立的 Spring 容器中，对一个特定的类创建了一个 bean，那么 Spring 容器通过 bean 的定义仅仅创建这个类的一个实例。在 Spring 中，单例范围是默认的范围。要在 XML 中定义单例的 bean，可以按照如下示例来编写：

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
<!-- 尽管是冗余的（单例范围是默认的），下面的bean定义和它是等价的 -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

4.5.2 原型范围

非单例，原型范围的 bean 就是当每次对指定 bean 进行请求时，一个新的 bean 的实例 就会创建。也就是说，bean 被注入到其它 bean 或是在容器中通过 getBean()方法来请求时就 会创建新的 bean 实例。作为一项规则，对所有有状态的 bean 使用原型范围，而对于无状 态的 bean 使用单例范围。

下图讲述了 Spring 的原型范围。数据访问对象（DAO）通常不是配置成原型的，因为典型的 DAO 不会持有任何对话状态；仅仅是对作者简单对单例图的重用。



下面的示例在 XML 中定义了原型范围的 bean：

```
<!-- 使用spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

和其它范围相比，Spring 不管理原型 bean 的完整生命周期；容器实例化，配置并装配 原型对象，并把他给客户端，对于原型 bean 的实例来说，就没有进一步的跟踪记录了。因 此，尽管初始化生命周期回调方法可以对所有对象调用而不管范围，那么在原型范围中，配 置销毁生命周期回调是不能被调用的。客户端代码必须清理原型范围的对象并且释放原型 bean 持有的系统资源。要让 Spring 容器来释放原型范围 bean 持有的资源，可以使用自定义 bean 后处理器（4.8.1 节），它持有需要被清理的 bean 的引用。

在某些方面，关于原型范围 bean 的 Spring 容器的角色就是对 Java new 操作符的替代。 所有之后生命周期的管理必须由客户端来处理。（关于 Spring 容器中 bean 的生命周期的详细内容，可以参考 4.6.1 节，“生命周期回调”）

4.5.3 单例 bean 和原型 bean 依赖

当你使用单例范围的 bean 和其依赖是原型范围的 bean 时，要当心依赖实在实例化时 被解析的。因此，如果依赖注入了原型范围的 bean 到单例范围的 bean 中，新的原型 bean 就被实例化并且依赖注入到单例 bean 中。原型实例是唯一的实例并不断提供给单例范围的 bean。

假设你想单例范围的 bean 在运行时可以重复获得新的原型范围 bean 的实例。那么就 不能将原型范围的 bean 注入到单例 bea n 中，因为这个注入仅仅发生一次，就是在 Spring 容器实例化单例 bean 并解析和注入它的依赖时。如果你在运行时需要原型 bean 新的实例 而不是仅仅一次，请参考 4.4.6 节，“方法注入”。

4.5.4 请求，会话和全局会话范围

request，session 和 global session 范围仅仅当你使用感知 Web 的 Spring ApplicationContext 实现类可用（比如 XmlWebApplicationContext）。如果你在 常规的 Spring IoC 容器中使用如 ClassPathXmlApplicationContext 这些范围，那么 就会因为一个未知的 bean 的范围而得到 IllegalStateException 异常。

4.5.4.1 初始化 Web 配置

要支持 request，session 和 global session 级别（Web 范围的 bean）范围的 bean，一些细微的初始化配置就需要在定义 bean 之前来做。（这些初始化的设置对标准的 范围，单例和原型，是不需要的。）

如果来完成这些初始化设置是基于你使用的特定 Servlet 容器的。

如果使用 Spring 的 Web MVC 来访问这些范围的 bean，实际上，是由 Spring 的 DispatcherServlet 或 DispatcherPortlet 来处理请求的，那就没有特殊的设置了：DispatcherServlet 和 DispatcherPortlet 已经可以访问所有相关的状态。

如果你使用支持 Servlet 2.4 规范以上的 Web 容器，在 Spring 的 DispatcherServle（比如，当 使 用 JSF 或 Struts 时 ） 之 外 来 处 理 请 求 ， 那 么 就 需 要 添 加 javax.servlet.ServletRequestListener 到 Web 应用的 web.xml 文件的声明中：

```
<web-app>
    ...
    <listener>
        <listener-class> org.springframework.web.context.request.RequestContextLi
            stener
        </listener-class>
    </listener>
    ...
</web-app>
```

如果你使用老版的 Web 容器（ Serlvet 2.3 ）， 那 么 就 使 用 提 供 的 javax.servlet.Filter 实 现 类 。 如 果 你 想 在 Servlet 2.3 容 器 中 ， 在 Spring 的 DispatcherServlet 外部的请求中访问 Web 范围的 bean，下面的 XML 配置代码片段就必须包 含在 Web 应用程序的 web.xml 文件中。（过滤器映射基于所处的 Web 应用程序配置，所以 你必须以适当的形式来修改它。）

```
<web-app>
    ..
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestCont extFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

DispatcherServlet，RequestContextListener 和 RequestContextFilter 所有都可以做相同的事情，即绑定 HTTP 请求对象到服务于请求的 Thread 中。这使得 bean 在请求和会话范围内对后续的调用链可用。

4.5.4.2 请求范围

考虑一下下面的 bean 定义：

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

Spring 容器通过使用 LoginAction bean 的定义来为每个 HTTP 请求创建 LoginAction bean 的新的实例。也就是说，LoginAction bean 是在 HTTP 请求级别的范 围。你可以改变创建的实例内部的状态，怎么改都可以，因为从相同的 LoginAction bean 定义中创建的其它实例在状态上不会看到这些改变；它们对单独的请求都是独立的。当请求 完成处理，bean 在请求范围内被丢弃了。

4.5.4.3 会话范围

考虑一下下面的 bean 定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Spring 容器通过使用 UserPreferences bean 的定义来为每个 HTTP 会话的生命周期 创建 UserPreferences bean 的新的实例。换句话说，UserPreferences bean 在 HTTP

session 级别的范围内是有效的。正如 request-scoped 的 bean，你可以改变创建的实 例 内 部 的 状 态 ， 怎 么 改 都 可以 ， 要 知 道其 它 HTTP session 实例也是使用从相同 UserPreferences bean 的定义中创建的实例，它们不会看到状态的改变，因为它们对单 独的 HTTP session 都是独立的。当 HTTP session 最终被丢弃时，那么和这个 HTTP session 相关的 bean 也就被丢弃了。

4.5.4.4 全局会话范围

考虑一下下面的 bean 定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session 范围和标准的 HTTP Session 范围（上面 4.5.4.3 节讨论的）类似， 但仅能应用于基于 portlet 上下文的 Web 应用程序。portlet 规范定义了全局的 Session 的 概念，该会话在所有 portlet 之间所共享来构建一个单独的 portlet Web 应用程序。在 global session 范围内定义的 bean 的范围（约束）就会在全局 portlet Session 的生命周期内。

如果你编写了一个标准的基于 Servlet 的 Web 应用，并且定义了一个或多个有 global session 范围的 bean，那么就会使用标准的 HTTP Session 范围，也不会抛出什么错误。

4.5.4.5 各种范围的 bean 作为依赖

Spring 的 IoC 容器不仅仅管理对象（bean）的实例，而且也装配协作者（依赖）。如果 你想注入（比如）一个 HTTP 请求范围的 bean 到另外一个 bean 中，那么就必须在有范围 bean 中注入一个 AOP 代理。也就是说，你需要注入一个代理对象来公开相同的公共接口作为有 范围的对象，这个对象也可以从相关的范围（比如，HTTP 请求范围）中获取真实的目标对 象并且委托方法调用到真正的对象上。

> 注意
>
> 你不需要使用 `<aop:scoped-proxy/>` 来结合 singleton 和 prototype 范围的 bean 。 如 果 你 想 为 单 例 bean 尝 试 创 建 有 范 围 的 代 理 ， 那 么 就 会 抛 出 BeanCreationException 异常。

在下面的示例中，配置仅仅只有一行，但是这对理解“为什么”和其中的“怎么做”是至关重要的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <!-- HTTP 会话范围的bean作为代理 -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
        <!-- 这个元素影响周围bean的代理 -->
        <aop:scoped-proxy/>
    </bean>
    <!-- 单例范围的bean用上面的代理bean来注入 -->
    <bean id="userService" class="com.foo.SimpleUserService">
        <!-- 作为代理的userPreferences bean引用 -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

要创建这样的代理，就要插入一个子元素 `<aop:scoped-proxy/>` 到有范围的 bean 的 定义中。（如果要选择基于类的代理，那么需要在类路径下放置 CGLIB 的类库，参考下面的 “选择创建代理类型”部分和“附录 C，基于 XML Schema 的配置”）为什么 request， session， globalSession 和自定义范围级别的 bean 需要 `<aop:scoped-proxy/>` 元 素？我们来看看下面的单例 bean 定义，并将它和你需要定义的上述范围来比较一下。（下面 的 userPreferences bean 的定义是不完整的。）

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

在上面 的例 子中， 单 例 bean userManager 是用 HTTP session 范围 的 bean userPreferences 的引用注入的。这里突出的一点就是 userManager bean 是单例的； 那么对于每个容器来说，它只会被实例化一次，并且它的依赖（本例中只有一个依赖，就是 userPreferences bean）也只会被注入一次。这就意味着 userManager bean 只会对同 一个 userPreferences 对象进行操作，也就是说，是最初被注入的那一个对象。

这并不是你想要的行为，即注入生命周期范围短的 bea n 到生命周期范围长的 bea n 中， 比如注入 HTTP session 范围的协作 bean 作为依赖到单例 bean 中。相反，你需要一个单独的 userManager 对 象， 生 命 周 期 和 HTTP session 一 样， 你 需 要 一 个 userPreferences 对象来特定于 HTTP session。因此容器创建对象，公开相同的公共 接口，而 UserPreferences 类（理想情况下是 UserPreferences 实例的对象）从范围机制（HTTP 请求，session 等等）来获取真正的 UserPreferences 对象。容器注入这 个代理对象到 userManager bean 中，这个 bean 不能感知 UserPreferences 的引用就 是代理。在这个示例中，当 UserManager 实例调用依赖注入的 UserPreferences 对象 的方法时，那么其实是在代理上调用这个方法。然后代理从（在本例中）HTTP session 中 获取真实的 UserPreferences 对象， 并且委托方法调用到获取的真实 的 UserPreferences 对象中。

因此，当注入 request 范围，session 范围和 globalSession 范围的 bean 到协作 对象时，你需要下面的，正确的，完整的配置：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

选择创建代理类型

默认情况下，当 Spring 容器为被 `<aop:scoped-proxy/>` 元素标记的 bean 创建代理时， 基于 CGLIB 的类的代理就被创建了。这就是说在应用程序的类路径下需要有 CGLIB 的类库。 注意：CGLIB 代理仅仅拦截公有的方法调用！不会调用代理中的非公有方法；它们不会被委托到有范围的目标对象。

另外，你可以配置 Spring 容器有范围的 bean 创建标准的基于 JDK 接口的代理，通过指定 `<aop:scoped-proxy/>` 元素的 proxy-target-class 属性值为 false。使用基于 JDK 接口的代理意味着你不需要在应用程序的类路径下添加额外的类库来使代理生效。然而， 它也意味着有范围 bean 的类必须实现至少一个接口，并且注入到有范围 bean 的所有协作 者必须通过它的一个接口来引用 bean。

```
<!—DefaultUserPreferences实现了UserPreferences接口 -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

关于选择基于类或基于接口的代理的更多详细信息，可以参考 8.6 节，“代理机制”。

4.5.5 自定义范围

在 Spring 2.0 当中，bean 的范围机制是可扩展的。你可以定义自己的范围，或者重新定 义已有 的范 围， 尽管 后者 被认 为是 不良 实践 而且 你不能 覆盖 内建 的 singleton 和 prototype 范围。

4.5.5.1 创建自定义范围

要 整 合 自 定 义 范 围 到 Spring 的 容 器 中 ， 那 么 需 要 实 现 org.springframework.beans.factory.config.Scope 接口，这会在本节中来介绍。 对于如何实现你自己的范围的想法，可以参考 Spring Framework 本身提供的 Scope 实现和 Scope 的 JavaDoc文档，这里会介绍你需要实现的方法的更多细节。

Scope 接口有四个方法来从作用域范围中获取对象，从作用域范围中移除对象还有允 许它们被销毁。

下面的方法从底层范围中返回对象。比如，会话范围的实现，返回会话范围的 bean（如 果它不存在，在绑定到会话后为将来的引用，这个方法返回 bean 的新的实例）。

```
Object get(String name, ObjectFactory objectFactory)
```

下面的方法从底层范围中移除对象。比如会话范围的实现，从底层的会话中移除会话范 围的 bean。对象应该被返回，但是如果特定名字的对象没有被发现的话可以返回 null。

```
Object remove(String name)
```

下面的方法注册当被销毁或者指定范围内的对象被销毁时，相关范围应用执行的回调函 数。参考 JavaDoc 文档或者 Spring 范围实现来获取更多关于销毁回调的信息。

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

下面的方法得到一个底层范围的会话标识符。这个标识符对每种范围都是不同的。对于 会话范围的实现，这个标识符可以是会话标识符。

```
String getConversationId()
```

4.5.5.2 使用自定义范围

在你编写并测试一个或多个自定义 Scope 实现之后，你需要让 Spring 容器感知你的新 的范围。下面的方法是用 Spring 容器注册新的 Scope 的核心方法。

```
void registerScope(String scopeName, Scope scope);
```

这个方法在 ConfigurableBeanFactory 接口中声明，也在 大多数的 ApplicationContext 实现类中可用，通过 BeanFactory 属性跟着 Spring。

registerScope(..)方法的第一个参数是和范围相关的唯一名称；在 Spring 容器本 身这样的名字就是 singleton 和 prototype。registerScope(..)方法的第二个参数 是你想注册并使用的自定义 Scope 实现的真实实例。

假设你编写了自定义的 Scope 实现，之后便如下注册了它。

> 注意
>
> 下面的示例使用了 SimpleThreadScope，这是包含在 Spring 当中的，但模式是没有 注册的。这些指令和你自己自定义的 Scope 的实现是相同的。

```
Scope threadScope = new SimpleThreadScope(); beanFactory.registerScope("thread", threadSc
```

之后就可以为你的自定义 Scope 创建符合 Spring 规则的 bean 的定义：

```
<bean id="..." class="..." scope="thread">
```

使用自定义的 Scope 实现，并没有被限制于程序注册。你也可以进行声明式的 Scope注册，使用 CustomScopeConfigurer 类：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/sprin
    <bean
        class="org.springframework.beans.factory.config.CustomScopeC onfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.support.SimpleT hreadScope"/
                </entry>
            </map>
        </property>
    </bean>
    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>
    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>
</beans>
```

> 注意
>
> 当在 FactoryBean 的实现中放置`<aop:scoped-proxy/>`时，那就是工厂 bean 本身被放 置到范围中，而不是从 getObject()方法返回的对象。

4.6 自定义 bean 的性质

4.6.1 生命周期回调

为了和容器管理 bean 的生命周期进行交互，你可以实现 Spring 的 InitializingBean 和 DisposableBean 接口。容器为前者调用 afterPropertiesSet()方法，为后者调 用 destroy()方法在 bean 的初始化和销毁阶段允许 bean 执行特定的动作。你也可以和容 器达到相同的整合，而不需要通过使用初始化方法和销毁方法的对象定义元数据来耦合你的类到 Spring 的接口中。

从内部来说，Spring Framework 使用了 BeanPostProcessor 的实现来处理任何回调 接口，它可以发现并调用合适的方法。如果你需要自定义特性或其它生命周期行为，Spring 不会提供开箱，你可以自行实现 BeanPostProcessor 接口。要获取更多信息，可以参考 4.8 节，“容器扩展点”。

除了初始化和销毁回调，Spring 管理的对象也会实现 Lifecycle 接口，所以那些对象 可以参与到启动和关闭过程，来作为容器自己的生命周期。

生命周期回调接口在本节中来说明。

4.6.1.1 初始化回调

org.springframework.beans.factory.InitializingBean 接口允许 bean 在 所有必须的属性被容器设置完成之后来执行初始化工作。InitializingBean 接口中仅仅 指定了一个方法：

```
void afterPropertiesSet() throws Exception;
```

推荐不要使用 InitializingBean 接口，因为它不必要地将代码耦合到 Spring 中。 另外，可以指定一个 POJO 初始化方法。在基于 XML 配置元数据的情形中，你可以使用 init-method 属性来指定签名为无返回值，无参数的方法的名称。比如，下面的定义：

```
<bean id="exampleInitBean"
    class="examples.ExampleBean"
    init-method="init"/>
public class ExampleBean {
    public void init() {
        // 做一些初始化工作
    }
}
```

这和下面的做法是完全相同的：

```
<bean id="exampleInitBean"
    class="examples.AnotherExampleBean"/>
public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        //做一些初始化工作
    }
}
```

但是没有耦合任何代码到 Spring 中。

4.6.1.2 销毁回调

实现 org.springframework.beans.factory.DisposableBean 接口，当容器 包含的 bean 被销毁时，允许它获得回调。DisposableBean 接口中仅仅有一个方法：

```
void destroy() throws Exception;
```

推荐不要使用 DisposableBean 接口，因为它不必要地将代码耦合到 Spring 中。另外， 可以指定一个由 bean 支持的通用的方法。使用基于 XML 配置元数据，你可以使用 `<bean/>` 元素中的 destroy-method 属性。比如，下面的定义：

```
<bean id="exampleInitBean"
    class="examples.ExampleBean"
    destroy-method="cleanup"/>
public class ExampleBean {
    public void cleanup() {
        // 做一些销毁工作 (比如释放连接池的连接)
    }
}
```

这和下面的做法是完全相同的：

```
<bean id="exampleInitBean"
    class="examples.AnotherExampleBean"/>
public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // 做一些销毁工作 (比如释放连接池的连接)
    }
}
```

但是没有耦合代码到 Spring 中。

4.6.1.3 默认的初始化和销毁方法

当使用 Spring 指定的 InitializingBean 和 DisposableBean 回调接口来编写初 始化和销毁方法回调时，典型地做法就是编写名称为 init()，initialize()，dispose() 等方法.理想的情况，这样的生命周期回调方法的名称在一个项目中是标准化的，那么所有 的开发人员可以使用相同的方法名称来保证一致性。

你可以配置 Spring 容器在每个 bean 中来查看命名的初始化和销毁回调方法名称。这就 是说，作为应用程序开发人员，你可以编写应用类并且使用初始化回调方法 init()，而不 需要在每个 bean 的定义中来配置 init-method="init"属性。当 bean 被创建时（并按 照前面描述的标准的生命周期回调合约），Spring 的 IoC 容器调用这个方法。这个特性也对 初始化和销毁方法回调强制执行了一致的命名约定。

假设你的初始化回调方法命名为 init()，销毁回调方法命名为 destroy()。那么你 就可以按照如下示例来装配类：

```
public class DefaultBlogService implements BlogService {
    private BlogDao blogDao;
    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }
    // 这是（毋庸置疑的）初始化回调方法
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }
}
<beans default-init-method="init">
    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>
</beans>
```

顶层 `<beans/>` 元素的 default-init-method 属性的存在，就会让 Spring 的 IoC 容 器意识到在 bean 中的一个叫做 init 的方法就是初始化回调方法。当一个 bean 被创建和装 配时，如果 bean 类有这样一个方法，那么它就会在合适的时间被调用。

相似地（也就是在 XML 中），你可以使用顶层元素 `<beans/>` 的 default-destroy-method 属性来配置销毁回调方法。

在已经存在的 bean 类中，有命名差异的回调方法，你可以指定（也就是在 XML 中） `<bean/>` 本身的 init-method 和 destroy-method 属性来覆盖默认的方法。

Spring 容器保证了在给 bean 提供所有的依赖后，配置的初始化回调方法会立即被调用。 因此初始化回调也被称为原始的 bean 引用，也就意味着 AOP 的拦截器等尚未应用到 bean 中。目标 bean 首先被完全创建，之后 AOP 代理（比方说）和它的拦截器链就会被应用上。

如果目标 bean 和代理分开来定义了，你的代码仍然可以绕开代理和原始目标 bea n 来交互。因此，引用拦截器到初始化方法中会是不一致的，因为这么来做了会和它的代理/拦截耦合 目标 bean 的合生命周期，当你的代码直接和原始目标 bean 交互时，会留下奇怪的语义。

4.6.1.4 组合生命周期机制

在 Spring 2.5 中，对控制 bean 的生命周期行为有三种选择：InitializingBean（4.6.1.1节）和 DisposableBean（4.6.1.2 节）回调接口；自定义 init()和 destroy()方法；@PostConstruct 和 @PreDestroy 注解（4.9.6 节）。你可以组合这些机制来控制给定的 bean。

> 注意
>
> 如果对一个 bean 配置了多个生命周期机制，并且每种机制都用不同的方法名称来配置，那么每个配置方法就会以下面列出的顺序来执行。然而，如果配置了相同的方法名称，比如说，init()对于初始化方法，多于一个生命周期机制，那么这个方法就执行一次，这在前 面章节解释过了。

相同的 bean 配置了不同初始化方法的多个生命周期机制，那么就按下面顺序调用：

- 使用@PostConstruct 注解的方法
- 由 InitializingBean 回调接口定义的 afterPropertiesSet()方法

自定义配置的 init()方法 销毁方法也是同样顺序调用：

- 使用@PreDestroy 注解的方法
- 由 DisposableBean 回调接口定义的 destroy ()方法
- 自定义配置的 destroy ()方法

4.6.1.5 启动和关闭回调

Lifecycle 接口定义了对于任意有它自己生命周期需求（比如，开始和结束一些后台 进程）对象的必要方法：

```
public interface Lifecycle {
    void start();
    void stop();
    boolean isRunning();
}
```

任何 Spring 管理的对象可以实现这个接口。那么，当 ApplicationContext 本身启动和关 闭时，它会级联那些定义在上下文中，对所有生命周期实现的调用。它会委托给 LifecycleProcessor 来做：

```
public interface LifecycleProcessor extends Lifecycle {
    void onRefresh();
    void onClose();
}
```

要注意 LifecycleProcessor 本身是扩展了 Lifecycle 接口。而且它添加了两个 方法来对上下文的刷新和关闭做出反应。

启动和关闭调用的顺序是很重要的。如果一个“依赖”关系存在于两个任意对象间，那 么需要依赖的一方会在它的依赖启动之后启动，并且在它的依赖关闭之前停止。然而，有时 直接的依赖关系是未知的。你可能仅仅知道确定类型的对象应该优先于另外一种类型对象的启动。在那些情形下，SmartLifecycle 接口定义了另外一种选择，命名为 getPhase() 方法，在它的父接口 Phased 中来定义。

```
public interface Phased {
    int getPhase();
}
public interface SmartLifecycle extends Lifecycle, Phased {
    boolean isAutoStartup();
    void stop(Runnable callback);
}
```

当启动时，最低层的对象首先启动，当停止时，是相反的。因此，实现了 SmartLifecycle 接口，并且 getPhase()方法返回 Integer.MIN_VALUE 的对象就会在最 先开始和最后停止的中间。在范围的另外一端，Integer.MIN_VALUE 的层次值表示对象 应该最后启动并且最先停止（可能是因为它取决于其它进程的执行）。当考虑层次值时，知道任意“普通”对象并且没有实现 Lifecycle 接口的默认的层次是 0，这也是很重要的。 因此，任意负的层次值就表示对象应该在那些标准组件之前启动（在它们之后停止），对于 任意正的层次值，反之亦然。

正如你看到由 SmartLifecycle 定义的停止方法接收回调。任何实现类必须在实现类关闭进程完成之后，调用回调方法 run()。这使得必要时可以进行异步关闭，因为默认的 LifecycleProcessor 接口的实现，DefaultLifecycleProcessor,，会等待每个阶 段一组对象来调用回调的超时值。每个阶段的默认超时是 30 秒。你可以在上下文中定义名为“lifecycleProcessor”的 bean 来覆盖默认的生命周期进程实例。如果你仅仅想去修改超时 时间，那么定义下面的 bean 就足够了：

```
<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycle
    <!-- 毫秒数的超时时间 -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

正如所提到的，LifecycleProcessor 接口定义了刷新和关闭上下文的回调方法。后 者将驱动关闭进程，就好像明确地调用了 stop()方法，但是当上下文关闭时它才会发生。另 一方面，‘刷新’回调使得 SmartLifecycle bean 的另外一个特性可用。当上下文刷新时（在所有对象都被实例化和初始化后），才会调用回调方法，在那时，默认的生命周期进程 会检查每个 SmartLifecycle 对象的 isAutoStartup()方法返回的布尔值。如果是“true”，那时对象将会被启动，而不是等待明确的上下文调用或它自己的 start()方法（不像上下文刷新，对标准的上下文实现来说，上下文启动不会自动发生）。“阶段”值和任意“依 赖”关系会按照上面所描述的相同方式来决定启动的顺序。

4.6.1.6 在非 Web 应用中，优雅地关闭 Spring IoC 容器

> 注意
>
> 本节仅对非 Web 应用来说。在相关 Web 应用程序关闭时，Spring 的基于 Web 的 ApplicationContext 实现已经有代码来优雅地关闭 Spring 的 Ioc 容器了。

如果你在非 Web 应用环境中使用 Spring 的 IoC 容器；比如，在富客户端桌面环境中； 你在 JVM 中注册了一个关闭的钩子。这么来做就保证了优雅地关闭并且在单例 bean 中调用相关销毁方法，那么所有资源都会被释放。当然，你必须正确配置并实现这些销毁回调方法。 要注册 一个 关闭 钩子， 可以 调用 AbstractApplicationContext 类中 声明的 registerShutdownHook()方法：

```
import org.springframework.context.support.AbstractApplicationContext; import org.springf
public final class Boot {
        public static void main(final String[] args) throws Exception { AbstractApplicati
            = new ClassPathXmlApplicationContext(new String []{"beans.xml"});
        // 对上面的上下文添加一个关闭的钩子...
        ctx.registerShutdownHook();
        // 这里运行应用程序...
        // 主方法退出,钩子在应用关闭前优先被调用...
    }
}
```

4.6.2 ApplicationContextAware 和 BeanNameAware

当 ApplicationContext 创 建 实 现 了 org.springframework.context.ApplicationContextAware 接口的类时，这个类 就被提供了一个 ApplicationContext 的引用。

```
public interface ApplicationContextAware {
    void setApplicationContext(ApplicationContext applicationContext) throws BeansExcepti
}
```

因此 bean 就可以编程来操作创建 它们的 ApplicationContext， 通过 ApplicationContext 接口，或者转换到这个接口的已知子 类型，比如 ConfigurableApplicationContext，会公开更多的功能。一种使用方式就是编程式获 取其它的 bean。有时候这种能力是很有用的。然而，通常你应该避免这么来做，因为它会耦合代码到 Spring 中，还没有协作者作为属性提供给 bean 的控制反转的风格。应用上下文 的其它方法提供了访问文件资源，公共应用事件和访问消息资源的方法。这些补充特性为在 4.14 节，“应用上下文的补充功能”来说明。

在 Spring 2.5 中，自动装配是获取 ApplicationContext 引用的另外一种方式。“传 统的”constructor 和 byType 自动装配模式（在 4.4.5 节，“自动装配协作者”中说明的） 可以分别为构造方法参数或 setter 方法参数提供 ApplicationContext 类型的依赖。为了更大的灵活性，包括自动装配字段和多参数方法的能力，还有使用新的基于注解的自动装 配特性。如果字段，构造方法或者普通方法进行@Autowired 注解，而且你这么做了， ApplicationFactory 就会自动装配到期望 BeanFactory 类型的字段，构造方法参数或方法参数。要获取更多信息，可以参考 4.9.2 节，“@Autowired 和@Inject”。

当 应 用 上 下 文 创 建 实 现 了 org.springframework.beans.factory.BeanNameAware 接口的类时，这个类会被 提供一个在相关对象中定义的命名引用。

```
public interface BeanNameAware {
    void setBeanName(string name) throws BeansException;
}
```

回调函数在普通 bean 的属性被填充之后并且在如 InitializingBean 的 afterPropertiesSet 或自定义初始化方法的初始化回调之前被调用。

4.6.3 其它 Aware 接口

除了上面讨论的 ApplicationContextAware 和 BeanNameAware，Spring 还提供了很多 Aware 接口，它们允许 bean 来告诉容器它们需要确定的基础的依赖。最重要的 Aware 接口在下面总结出来了-作为一个通用的规则，名称就很好地说明了依赖的类型：

表 4.4 Aware 接口

名称	注入的依赖	何处解释
ApplicationContextA ware	声明 ApplicationContext	4.6.2 节，“ ApplicationContextAware 和 BeanNameAware”
ApplicationEventPub lisherAware	包含在 ApplicationContext内的事件发布	4.14 节，“应用上下文的补充 功能”
BeanClassLoaderAware	用于加载 bean 类的类加载器	4.3.2 节，“实例化 bean”
BeanFactoryAware	声明 BeanFactory	4.6.2 节，“ ApplicationContextA ware 和 BeanNameAware”
BeanNameAware	声明 bean 的名称	4.6.2 节，“ ApplicationContextA ware 和 BeanNameAware”
BootstrapContextAwa re	容器运行的资源适配器 BootstrapContext。典型地是仅仅在 JCA 感知的 ApplicationContext 中可用。	第 24 章，JCA CCI
LoadTimeWeaverAware	在加载时为处理类定义的织入器	8.8.4 节，“Spring Framework中使用 Aspect J 进行加载时织入”
MessageSourceAware	解析消息配置的策略（支持参数化和国际化）	4.14 节，“应用上下文的补充功能”
NotificationPublisherAware	Spring JMX 通知发布	23.7 节，“通知”

PortletConfigAware	容器运行的当前的 PortletConfig。仅在感知 Web 的 Spring 的 ApplicationContext 中有效	第 19 章，Portlet MVC 框架
PortletContextAware	容器运行的当前的 PortletContext。仅在感知 Web 的 Spring 的 ApplicationContext 中有效	第 19 章，Portlet MVC 框架
ResourceLoaderAware	为低级别的资源访问配置的加载器	第 5 章，资源
ServletConfigAware	容器运行的当前的 ServletConfig。仅在感知 Web 的 Spring 的 ApplicationContext 中有效	第 16 章，Web MVC 框架
ServletContextAware	容器运行的当前的 ServletContext。仅在感知 Web 的 Spring 的 ApplicationContext 中有效	第 16 章，Web MVC 框架

请再次注意绑定你的代码到 Spring API 的这些接口的使用，并不再遵循控制反转风格。

因此，它们被推荐使用到基础的 bean 中，那些 bean 需要编程式地访问容器。

4.7 Bean 定义的继承

bean 的定义可以包含很多配置信息包括构造方法参数，属性值和容器指定的信息，比 如初始化方法，静态工厂方法名称等。子 bean 定义继承从父 bean 中获得的配置元数据。 子 bean 可以覆盖一些值或者添加其它所需要的。使用父子 bean 定义可以节省很多输入。 实际上，这是一种模板形式。

如 果 你 编 程 式 地 使 用 ApplicationContext 接口，子 bean 的 定 义 可 以 由 ChildBeanDefinition 类代 表。 很多 用户 不使 用这 个级 别的 方法 ， 而 是在 类似于 ClassPathXmlApplicationContext.中声明式地配置 bean 的信息。当使用基于 XML 的配置元数据时，你可以使用 parent 属性来标识一个子 bean，使用这个属性的值来标识 父 bean。

```
<bean id="inheritedTestBean" abstract="true" class="org.springframework.beans.TestBean">
<property name="name" value="parent"/>
<property name="age" value="1"/>
</bean>
<bean id="inheritsWithDifferentClass" class="org.springframework.beans.DerivedTestBean" p
<property name="name" value="override"/>
<!-- age属性值1将会从父bean中继承过来 -->
</bean>
```

如果没有指定一个子 bean 使用父 bean 的类，但也可以覆盖它。在这种情形中，子 bean的类必须和父 bean 兼容，也就是说，它必须接受父 bean 的属性值。

子 bean 的定义继承构造方法参数值，属性值，还有父 bean 的方法覆盖，添加新值的 选择。任何你指定的初始化方法，销毁方法，和/或 static 工厂方法设置会覆盖对应父 bean 中的设置。

剩下的设置通常是从子 bean 来定义：依赖，自动装配模式，依赖检测，单例，范围， 延迟初始化。

前面的示例明确地使用了 abstract 属性来标识了父 bean 的定义。如果父 bean 没有 指定类，那么明确地标识父 bean 就必须要有 abstract，如下所示：

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>
<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean" parent="in
    init-method="initialize">
    <property name="name" value="override"/>
    <!-- age属性值1将会从父bean中继承过来 -->
</bean>
```

父 bean 不会被实例化，因为它自己是不完整的，而且也明确地被 abstract 标记。当 bean 的定义是 abstract 这样的，那么它也仅仅被用做纯 bean 定义的模板，作为子 bean 定义的父 bean。尝试使用这种自身是 abstract 的父 bean，作为另外一个 bean 参考的属 性来指代它，或者使用父 bean 的 id 来明确使用 getBean()方法调用，会返回错误。相似 地，容器内部的 preInstantiateSingletons()方法忽略了抽象 bean 的定义。

> 注意
>
> 默认情况下，预实例化所有单例 bean。因此，如果你有仅仅想用作模板的（父）bean，而且这个 bean 指定了一个类，那么必须将 abstract 属性设置为 true，这点是很重要的。否 则，应用上下文就会（尝试）预实例化 abstract 的 bean。

4.8 容器扩展点

通常情况下，应用程序开发人员不需要编写 ApplicationContext 实现类的子类。 相反，Spring 的 IoC 容器可以通过插件的方式来扩展，就是实现特定的整合接口。下面的几 个章节会来说明这些整合接口。

4.8.1 使用 BeanPostProcessor 来自定义 bean

BeanPostProcessor 接口定义了你可以提供实现你自己的（或覆盖容器默认的）实 例逻辑，依赖解析逻辑等的回调方法。如果你想在 Spring 容器完成实例化，配置和初始化 bean 之后实现一些自定义逻辑，那么你可以使用一个或多个 BeanPostProcessor 实现类 的插件。

你可以配置多个 BeanPostProcessor 实例，而且你还可以通过设置 order 属性来 控制这些 BeanPostProcessor 执行的顺序。 仅当 BeanPostProcessor 实现了 Ordered 接口你才可以设置设个属性；如果你想编写你自己的 BeanPostProcessor，你 也应该考虑实现 Ordered 接口。 要 了 解 更 多 细 节 ， 可 以 参 考 JavaDoc 文档中的 BeanPostProcessor 和 Ordered 接口。

> 注意
>
> BeanPostProcessor 操作 bean（对象）实例；那也就是说，Spring 的 IoC 容器 实例化 bean 的实例，之后 BeanPostProcessor 来做它们要做的事情。

BeanPostProcessor 的范围是对于每一个容器来说的。如果你使用了容器继承，那么这才有所关联。如果你在一个容器中定义了 BeanPostProcessor，那么它仅仅 会在那个容器中后处理 bean。换句话说，一个容器中定义的 bean 不会被另外一个容器 中定义的 BeanPostProcessor 来进行后处理，即便是两个容器都是相同继承链上的 一部分。

要修改真正的 bean 定义（也就是说，定义 bean 的蓝图），你可以使用 4.8.2 节，“使 用 BeanFactoryPostProcessor 来 自 定 义 配 置 元 数 据 ” 描 述 的 BeanFactoryPostProcessor 来进行。 org.springframework.beans.factory.config.BeanPostProcessor 接口由两个回调方法构成。如果在容器中有一个类注册为后处理器，对于容器创建的每个 bean 的实例，后处理器从容器中获得回调方法，在容器初始化方法之前（比如 InitializingBean 的 afterPropertiesSet()方法和任意声明为初始化的方法）被调用，还有在 bean 初始化回调之后被调用。后处理器可以对 bean 实例采取任何动作，包括完整忽略回调。通常来说 bean 的后处理器会对回调接口进行检查，或者会使用代理包装 bean。一些 Spring 的 AOP 基类也会 作为 bean 的后处理器实现来提供代理包装逻辑。

ApplicationContext 会自动检测任意实现了 BeanPostProcessor 接口的 bean 定义的配置元数据。ApplicationContext 注册这些 bean 作为后处理器，那么它们可以 在 bean 创建之后被调用。Bean 的后处理器可以在容器中部署，就像其它 bean 那样。

BeanPostProcessor 和 AOP 自动代理

实现了 BeanPostProcessor 接口的类是特殊的，会被容器不同对待。所有它们参照的 BeanPostProcessor 和 bea n 会在启动时被实例化，作为

ApplicationContext 启 动 阶 段 特殊 的 一 部 分 。 接 下 来 ， 所 有 的 BeanPostProcessor 以排序的方式注册并应用于容器中的其它 bean。因为 AOP 自动 代理作为 BeanPostProcessor 本身的实现，它们为自动代理资格的直接引用的既不是 BeanPostProcessor 也不是 bean，因此没有织入它们的方面。

对于这样的 bean，你应该看到一个信息级的日志消息："Bean foo 没有由所有 BeanPostProcessor 接口处理的资格（比如：没有自动代理的资格）"。 下面的示例展示了如何在 ApplicationContext 中编写，注册和使用BeanPostProcessor。

4.8.1.1 示例：BeanPostProcessor 风格的 Hello World

第一个示例说明了基本的用法。示例展示了一个自定义的 BeanPostProcessor 实现 类来调用每个由容器创建 bean 的 toString()方法并打印出字符串到系统的控制台中。

自定义 BeanPostProcessor 实现类的定义：

```
package scripting;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;
public class InstantiationTracingBeanPostProcessor implements
BeanPostProcessor {
    // 简单地返回实例化的bean
    public Object postProcessBeforeInitialization(Object bean, String beanName)throws Bea
        return bean; // 这里我们可能返回任意对象的引用...
    }
    public Object postProcessAfterInitialization(Object bean, String beanName)throws Bean
        System.out.println("Bean '" + beanName + "' created : " +
        bean.toString());
        return bean;
    }
}
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <lang:groovy id="messenger"
        script-source="classpath:org/springframework/scripting/gr oovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So
        Dreamy."/>
    </lang:groovy>
    <!--
    当上面的bean(messenger)被实例化时，这个自定义的BeanPostProcessor实 现会输出实际内容到系统控制台
    -->
    <bean
    class="scripting.InstantiationTracingBeanPostProcessor" />
</beans>
```

注意 InstantiationTracingBeanPostProcessor 仅仅是简单地定义。它也没有 命名，因为它会像其它 bean 那样被依赖注入。（前面的配置也可以定义成 Groovy 脚本支持 的 bean。Spring 2.0 动态语言支持在第 27 章中来详细说明）

下面示例的 Java 应用程序执行了前面配置的代码：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationConte xt;
import org.springframework.scripting.Messenger;
public final class Boot {
    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext("scripting/beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger"); System.out.println(me
    }
}
```

上面应用程序的输出类似于如下内容：

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961 or
```

4.8.1.2 示例：RequiredAnnotationBeanPostProcessor

配合自定义的 BeanPostProcessor 实现使用回调接口或者注解，是扩展 Spring IoC 容器的一种通用方式。Spring 的 RequiredAnnotationBeanPostProcessor 就是一个 例子 – 一种 BeanPostProcessor 实现，随着 Spring 一起发布，来保证 bean 中被（任意） 注解所标记的 JavaBean 属性在真正（配置）注入时有值。

4.8.2 使用 BeanFactoryPostProcessor 自定义配置元数据

下 一 个 扩 展 点 我 们 要 来 看 看 org.springframework.beans.factory.config.BeanFactoryPostProcessor 。 这个接口的 语义和 那 些 BeanPostProcessor 是相 似的， 但有一个 主要的不 同点：BeanFactoryPostProcessor 操作 bean 的配置元数据；也就是说 Spring 的 IoC 容器允许 BeanFactoryPostProcessor 来 读 取 配 置 元 数 据 并 在 容 器 实 例 化 BeanFactoryPostProcessor 以外的任何 bean 之前可以修改它。

你可以配置多个 BeanFactoryPostProcessor，并且你也可以通过 order 属性来控 制 这 些 BeanFactoryPostProcessor 执 行 的 顺 序 。 然 而 ， 仅 当 BeanFactoryPostProcessor 实现 Ordered 接口时你才能设置这个属性。如果编写你自己的 BeanFactoryPostProcessor，你也应该考虑实现 Ordered 接口。参考 JavaDoc 文档来获取 BeanFactoryPostProcessor 和 Ordered 接口的更多细节。

> 注意
>
> 如果你想改变真实的 bean 实例（也就是说，从配置元数据中创建的对象），那么你 需要使用 BeanPostProcessor（在上面 4.8.1 节，“使用 BeanPostProcessor 来自定义 bean ”中描述）来代替。在 BeanFactoryPostProcessor （比如使用 BeanFactory.getBean()）中来使用这些 bea n 的实例虽然在技术上是可行的，但 这么来做会引起 bean 过早实例化，违反标准的容器生命周期。这也会引发一些副作用， 比如绕过 bean 的后处理。

而且，BeanFactoryPostProcessor 的范围也是对每一个容器来说的。如果你 使 用 了容 器的继承 的话， 这 就是 唯一 相关 的点 了。 如果 你在 一个 容器 中定 义 了 BeanFactoryPostProcessor，那么它只会用于在那个容器中的 bean。一个容器中 Bean 的定义不会被另外一个容器中的 BeanFactoryPostProcessor 后处理，即便 两个容器都是相同继承关系的一部分。

当在 ApplicationContext 中声明时，bean 工厂后处理器会自动被执行，这就可以 对定义在容器中的配置元数据进行修改。Spring 包含了一些预定义的 bean 工厂后处理器，比如 PropertyOverrideConfigurer 和 PropertyPlaceholderConfigurer.。自定义的 BeanFactoryPostProcessor 也可以来用，比如，注册自定义的属性编辑器。

ApplicationContext 会 自 动 检 测 任 意 部 署 其 中 ， 且 实 现 了 BeanFactoryPostProcessor 接口的 bean。在适当的时间，它用这些 bean 作为 bean 工厂后处理器。你可以部署这些后处理器 bean 作为你想用的任意其它的 bean。

> 注意
>
> 和 BeanPostProcessor 一样，通常你不会想配置 BeanFactoryPostProcessor 来进行延迟初始化。如果没有其它 bean 引用 Bean(Factory)PostProcessor，那么后 处理器就不会被初始化了。因此，标记它为延迟初始化就会被忽略，即便你在 `<beans/>` 元 素声明中设置 default-lazy-init 属性为 true，那么 Bean(Factory)PostProcessor 也会正常被初始化。

4.8.2.1 示例：**PropertyPlaceholderConfigurer**

你可以使用来对使用了标准 Java Properties 格式的分离文件中定义的 bean 来声明属 性值。这么来做可以使得部署应用程序来自定义指定的环境属性，比如数据库的连接 URL 和密码，不会有修改容器的主 XML 定义文件或其它文件的复杂性和风险。

考虑一下下面这个基于 XML 的配置元数据代码片段，这里的 dataSource 就使用了占位符来定义。这个示例展示了从 Properties 文 件 中 配 置 属 性 的 方 法 。 在 运 行 时 ， PropertyPlaceholderConfigurer 就会用于元数据并为数据源替换一些属性。指定替 换的值作

为${属性-名}形式中的占位符，这里应用了 Ant/log4j/JSP EL 的风格。

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceho lderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>
<bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSour
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

而真正的值是来自于标准的 Java Properties 格式的文件：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

因此，字符串${jdbc.username}在运行时会被值'sa'替换，对于其它占位符来说也是相同 的，匹 配 到 了 属 性 文 件 中 的 键 就 会 用 其 值 替 换 占 位 符 。PropertyPlaceholderConfigurer 在很多 bean 定义的属性中检查占位符。此外，对 占位符可以自定义前缀和后缀。

使用 Spring 2.5 引入的 context 命名空间，也可以使用专用的配置元素来配置属性占 位符。在 location 属性中，可以提供一个或多个以逗号分隔的列表。

```
<context:property-placeholder
    location="classpath:com/foo/jdbc.properties"/>
```

PropertyPlaceholderConfigurer 不仅仅查看在 Properties 文件中指定的属 性。默认情况下，如果它不能在指定的属性文件中发现属性，它也会检查 Java System 属性。 你可以通过设置 systemPropertiesMode 属性，使用下面整数的三者之一来自定义这种 行为：

never(0)：从不检查系统属性

fallback(1)：如果没有在指定的属性文件中解析到属性，那么就检查系统属性。这是默 认的情况。

override(2)：在检查指定的属性文件之前，首先去检查系统属性。这就允许系统属性覆 盖其它任意的属性资源。

查看 PropertyPlaceholderConfigurer 的 JavaDoc 文档来获取更多信息。

类名替换

你可以使用 PropertyPlaceholderConfigurer 来替换类名，在运行时，当你不得不去选择一个特定的实现类时，这是很有用的。比如：

```
<bean class="org.springframework.beans.factory.config.PropertyPlac eholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/strategy.properties< /value>
    </property>
    <property name="properties">
        <value>custom.strategy.class=com.foo.DefaultStrategy< / value>
    </property>
</bean>
<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

如果类在运行时不能解析成一个有效的类，那么在即将创建时，bean 的解析就失 败 了， 这是 ApplicationContext 在 对 非 延 迟 初 始 化 bean 的 preInstantiateSingletons()阶段发生的。

4.8.2.2 示例：PropertyOverrideConfigurer

PropertyOverrideConfigurer， 另 外 一 种 bean 工 厂 后 处 理 器， 类 似 于 PropertyPlaceholderConfigurer，但不像后者，对于所有 bean 的属性，原始定义可 以有默认值或没有值。如果一个 Properties 覆盖文件没有特定 bean 的属性配置项，那么 就会使用默认的上下文定义。

注意，bean 定义是不知道被覆盖的，所以从 XML 定义文件中不能立即明显反应覆盖配 置。在多个 PropertyOverrideConfigurer 实例的情况下，为相同 bean 的属性定义不同的值，那么最后一个有效，这就是覆盖机制。 属性文件配置行像这种格式：

```
beanName.property=value
```

比如：

```
dataSource.driverClassName=com.mysql.jdbc.Driver dataSource.url=jdbc:mysql:mydb
```

这个示例文件可以用于包含了 dataSource bean 的容器，它有 driver 和 url 属性。 复合属性名也是支持的，除了最终的属性被覆盖，只要路径中的每个组件都是非空的（假设由构造方法初始化）。在这个例子中...

```
foo.fred.bob.sammy=123
```

...foo bean 的 fred 属性的 bob 属性的 sammy 属性的值设置为标量 123。 注意指定的覆盖值通常是文字值；它们不会被翻译成 bean 的引用。当 XML 中的 bean 定义的原始值指定了 bean 引用时，这个约定也适用。

使用 Spring 2.5 引入的 context 命名空间，可以使用专用的配置元素来配置属性覆盖：

```
<context:property-override
location="classpath:override.properties"/>
```

4.8.3 使用 FactoryBean 来自定义实例化逻辑

实现了 org.springframework.beans.factory.FactoryBean 接口的对象它们 就是自己的工厂。

FactoryBean 接口就是 Spring IoC 容器实例化逻辑的可插拔点。如果你的初始化代码 很复杂，那么相对于（潜在地）大量详细的 XML 而言，最好是使用 Java 语言来表达。你可 以 创建 自己 的 FactoryBean ，在类中编写复杂的初始化代码，之后将你自定义的 FactoryBean 插入到容器中。

FactoryBean 接口提供下面三个方法：

Object getObject()：返回工厂创建对象的实例。这个实例可能被共享，那就是看这个工厂返回的是单例还是原型实例了。

boolean isSingleton()：如果 FactoryBean 返回单例的实例，那么该方法返回 true， 否则就返回 false。

Class getObjectType()：返回由 getObject()方法返回的对象类型，或者事先不知 道类型时返回 null。

FactoryBean 的概念和接口被用于 Spring Framework 中的很多地方；随 Spring 发行， 有超过 50 个 FactoryBean 接口的实现类。

当你需要向容器请求一个真实的 FactoryBean 实例，而不是它生产的 bean，当调用 ApplicationContext 的 getBean()方法时，在 bean 的 id 之前要有连字符（&）。所以 对于一个给定 id 为 myBean 的 FactoryBean，调用容器的 getBean("myBean")方法返回的 FactoryBean 产品；而调用 getBean("&myBean")方法则返回 FactoryBean 实例本身。

4.9 基于注解的容器配置

配置Spring时，注解要比XML更好吗？ 基于注解配置的介绍就提出了这样的问题，这种方法要比XML‘更好’吗？简短的回答就是具体问题具体分析。完整的答案就是每种方法都有它的利与弊，通常是让开发 人员来决定使用哪种策略更适合使用。由于定义它们的方式，注解在它们的声明中提供了大量的上下文，使得配置更加简短和简洁。然而，XML更擅长装配组件，而不需要触碰它们源代码或重新编译。一些开发人员更喜欢装配源码而其他人认为被注解的类不再 是POJO了，此外，配置变得分散并且难以控制。

无论怎么去线则，Spring都可以容纳两种方式，甚至是它们的混合体。最值得指出 的是通过JavaConfig（4.12节）选择，Spring允许以非侵入式的方式来使用注解，而不需 要触碰目标组件的源代码和工具，所有的配置方式都是SpringSource Tool Suite所支持的。

作为 XML 配置的另外一种选择，依靠字节码元数据的基于注解的配置来装配组件代替 了尖括号式的声明。作为使用 XML 来表述 bean 装配的替换，开发人员可以将配置信息移入 到组件类本身中，在相关的类，方法或字段声明上使用注解。正如在 4.8.1.2 节，“示例：RequiredAnnotationBeanPostProcessor”所提到的，使用 BeanPostProcessor 来连接注解是扩展 Spring IoC 容器的一种常用方式。比如，Spring 2.0 引入的使用@Required（4.9.1 节）注解来强制所需属性的可能性。在 Spring 2.5 中，可以使用相同的处理方法来驱 动 Spring 的依赖注入。从本质上来说，@Autowired 注解提供了在 4.4.5 节，“自动装配协 作者”中描述的相同能力，但却有更细粒度的控制和更广泛的适用性。Spring 2.5 也添加了 对 JSR-250 注解的支持，比如@Resource，@PostConstruct 和@PreDestroy。Spring 3.0 添加了对 JSR-330（对 Java 的依赖注入）注解的支持，包含在 javax.inject 包下，比如@Inject，@Qualifier，@Named 和@Provider，当 JSR330 的 jar 包在类路径下时就可以使用。使 用这些注解也需要在 Spring 容器中注册特定的 BeanPostProcessor。

> 注意
>
> 注解注入会在 XML 注入之前执行，因此通过两种方式，那么后面的配置会覆盖前面装配的属性。

一如往常，你可以注册它们作为独立的 bean，但是它们也可以通过包含下面的基于 XML的 Spring 配置代码片段被隐式地注册（注意要包含 context 命名空间）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <context:annotation-config/>
</beans>
```

（隐式注册的后处理器包含 AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor， PersistenceAnnotationBeanPostProcessor， 以及上述的 RequiredAnnotationBeanPostProcessor）

> 注意
>
> `<context:annotation-config/>` 仅仅查找定义在同一上下文中的 bean 的注解。 这就意味着，如果你为 DispatcherServlet 将 `<context:annotation-config/>` 放 置在 WebApplicationContext 中，那么它仅仅检查控制器中的@Autowired bean，而 不是你的服务层 bean，可以参看 16.2 节，“DispatcherServlet”来查看更多信息。

4.9.1 @Required

@Required 注解应用于 bean 属性的 setter 方法，就向下面这个示例：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

这个注解只是表明受影响的 bean 的属性必须在 bean 的定义中或者是自动装配中通过明确的属性值在配置时来填充。如果受影响的 bean 属性没有被填充，那么容器就会抛出异 常；这就允许了急切而且明确的失败，要避免 NullPointerException。我们推荐你放 置断言到 bean 的类中，比如，放置到初始化方法中。当你在容器外部使用类时，这么来做 是强制那些所需的引用和值。

4.9.2 @Autowired 和@Inject

正如预期的那样，你可以使用@Autowired 注解到"传统的"setter 方法中： 注意在下面的示例中，JSR330 的@Inject 注解可以用于代替 Spring 的@Autowired。没有必须的属性，不像 Spring 的@Autowired 注解那样，如果要注入的值是可选的话，要有一个 required 属性来表示。如果你将 JSR330 的 JAR 包放置到类路径下的话，这种行为就会自 动开启。

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

你也可以将注解应用于任意名称和/或多个参数的方法：

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao customerPreferen
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

你也可以将用于构造方法和字段：

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

也可以提供 ApplicationContext 中特定类型的所有 bean，通过添加注解到期望哪 种类型的数组的字段或者方法上：

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog[] movieCatalogs;
    // ...
}
```

相同地，也可以用于特定类型的集合：

```
public class MovieRecommender {
    private Set<MovieCatalog> movieCatalogs;
    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }
    // ...
}
```

甚至特定类型的 Map 也可以自动装配，但是期望的键的类型是 String 的。Map 值会 包含所有期望类型的 bean，而键会包含对应 bean 的名字：

```
public class MovieRecommender {
    private Map<String, MovieCatalog> movieCatalogs;
    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }
    // ...
    }
```

默认情况下，当出现零个候选 bean 的时候，自动装配就会失败；默认的行为是将被注解的方法，构造方法和字段作为需要的依赖关系。这种行为也可以通过下面这样的做法来改变。

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired(required=false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

> 注意
>
> 每一个类中仅有一个被注解方法可以标记为必须的，但是多个非必须的构造方法可 以被注解。在那种情况下，每个构造方法都要考虑到而且 Spring 使用依赖可以被满足 的那个构造方法，那就是参数最多的那个构造方法。

@Autowired 需要的属性推荐使用@Required 注解。所需的表示了属性对于自动 装配目的不是必须的，如果它不能被自动装配，那么属性就会忽略了。另一方面，

@Required 更健壮一些，它强制了由容器支持的各种方式的属性设置。如果没有注入任何值，就会抛出对应的异常。 你也可以针对我们熟知的解决依赖关系的接口来使用 @Autowired：BeanFactory，ApplicationContext， Environment， ResourceLoader，ApplicationEventPublisher 和 MessageSource。这些接口和它们的扩展接口，比 如 ConfigurableApplicationContext 或 ResourcePatternResolver 也会被自动解析，而不需要特殊设置的必要。

```
public class MovieRecommender {
    @Autowired
    private ApplicationContext context;
    public MovieRecommender() {

    }
    // ...
}
```

> 注意
>
> @Autowired， @Inject， @Resource 和 @Value 注 解 是 由 Spring 的 BeanPostProcessor 实现类来控制的，反过来告诉你你不能在 BeanPostProcessor 或 BeanFactoryPostProcessor 类型（任意之一）应用这些注解。这些类型必须明确地 通过 XML 或使用 Spring 的@Bean 方法来'装配'。

4.9.3 使用限定符来微调基于注解的自动装配

因为通过类型的自动装配可能导致多个候选者，那么在选择过程中通常是需要更多的控 制的。达成这个目的的一种做法就是 Spring 的@Qualifier 注解。你可以用特定的参数来 关联限定符的值，缩小类型的集合匹配，那么特定的 bean 就为每一个参数来选择。最简单 的情形，这可以是普通描述性的值：

> 注意
>
> JSR 330 的@Qualifier 注解仅仅能作为元注解来用，而不像 Spring 的@Qualifier 可以使用字符串属性来在多个注入的候选者之间区别，并可以放在注解，类型，字段，方法，构 造方法和参数中。

```
public class MovieRecommender {
    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
    // ...
}
```

@Qualifier 注解也可以在独立构造方法参数或方法参数中来指定：

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,CustomerPreferenceDa
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

对应的 bean 的定义可以按如下所示。限定符值是“main”的 bean 会用限定了相同值 的构造方法参数来装配。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- 注入这个bean需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- 注入这个bean需要的任何依赖 -->
    </bean>
    <bean id="movieRecommender" class="example.MovieRecommender"/>
</beans>
```

对于后备匹配，bean 名称被认为是默认的限定符值。因此你可以使用 id 为“main”来 定义 bean，来替代嵌套的限定符元素，这也会达到相同的匹配结果。然而，尽管你使用这 种规约来通过名称参照特定的 bean，@Autowired 从根本上来说就是关于类型驱动注入和可 选语义限定符的。这就是说限定符的值，即便有 bean 的名称作为后备，通常在类型匹配时 也会缩小语义；它们不会在语义上表达对唯一 bean 的 id 的引用。好的限定符的值是“main” 或“EMEA”或“persistent”，特定组件的表达特性是和 bean 的 id 独立的，在比如前面示例 之一的匿名 bean 的情况下它是可能自动被创建的。

限定符也可以用于类型集合，正如上面讨论过的，比如 `Set<MovieCatalog>` 。在这 种情况下，根据声明的限定符，所有匹配的 bean 都会被注入到集合中。这就说明了限定符 不必是唯一的；它们只是构成了筛选条件。比如，你可以使用相同的限定符“action”来定 义多个 MovieCatalog bean；它们全部都会通过@Qualifier("action")注解注入到 `Set<MovieCatalog>` 中。

> 提示
>
> 如果你想通过名称来表达注解驱动注入，主要是不使用@Autowired，即便在技术上来说能够通过@Qualifier 值指向一个 bean 的名称。相反， 使用 JSR-250 的 @Resource 注解，在语义上定义了通过它的唯一名称去确定一个具体的目标组件，声 明的类型和匹配过程无关。

由于这种语义区别的特定后果，bean 本身被定义为集合或 map 类型，是不能通过 @Autowired 来进行注入的，因为类型匹配用于它们不太合适。对于这样的 bean 使用 @Resource，通过唯一的名称指向特定的集合或 map 类型的 bean。

@Autowired 可以用于字段，构造方法和多参数方法，在参数级允许通过限定符 注解缩小。相比之下，@Resource 仅支持字段和仅有一个参数的 bea n 属性的 setter 方法。因此，如果你的注入目标是构造方法或多参数的方法，那么就坚持使用限定符。 你可以创建你自定义的限定符注解。只需定义一个注解并在你的定义中提供 @Qualifier 注解即可。 注意你可以以下面描述的方式来使用 JSR 330 的@Qualifier 注解，用于替换 Spring 的 @Qualifier 注解。如果在类路径中有 JSR 330 的 jar 包，那么这种行为是自动开启的。

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre { String value(); }
```

之后你可以在自动装配的字段和参数上提供自定义的限定符：

```
public class MovieRecommender {
    @Autowired
    @Genre("Action")
    private MovieCatalog actionCatalog;
    private MovieCatalog comedyCatalog;
    @Autowired
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
        this.comedyCatalog = comedyCatalog;
    }
    // ...
}
```

之后，为候选 bean 提供信息，你可以添加 `<qualifier/>` 标签来作为 `<bean/>` 标签的 子元素，然后指定 type 和 value 值来匹配你自定义的限定符注解。这种类型匹配是基于 注解类的完全限定名。否则，作为一种简便的形式，如果没有名称冲突存在的风险，你可以 使用短类名。这两种方法都会在下面的示例中来展示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
http://www.springframework.org/schema/context http://www.springframework.org/schema/conte
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="Genre" value="Action"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="example.Genre" value="Comedy"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean id="movieRecommender" class="example.MovieRecommender"/>
</beans>
```

在 4.10 节，“类路径扫描和管理的组件”中，你会看到基于注解的替代，在 XML 中来 提供限定符元数据。特别是在 4.10.7 节，“使用注解提供限定符元数据”中。

在一些示例中，使用无值的注解就足够了。这当注解服务于多个通用目的时是很有用的，

而且也可以用于集中不同类型的依赖关系。比如，当没有因特网连接时，你可以提供脱机目录来由于搜索。首先定义简单的注解：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {
}
```

之后将注解添加到要被自动装配的字段或属性上：

```
public class MovieRecommender {
    @Autowired
    @Offline
    private MovieCatalog offlineCatalog;
    // ...
}
```

现在来定义 bean 的限定符的 type：

```
<bean class="example.SimpleMovieCatalog">
<qualifier type="Offline"/>
<!— 注入这个bean所需要的任何依赖 -->
</bean>
```

你也可以定义自定义的限定符注解来接受命名属性，除了或替代简单的 value 属性。 如果多个属性值之后在要不自动装配的字段或参数上来指定，那么要考虑 bean 的定义必须 匹配自动装备候选者的所有属性值。示例，考虑下面的注解定义：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier { String genre();
Format format();
}
```

在本例中，Format 是枚举类型：

```
public enum Format { VHS, DVD, BLURAY }
```

要自动装配的字段使用自定义的限定符和包含 genre 和 format 两个属性的值来注解。

```
public class MovieRecommender {
    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Action")
    private MovieCatalog actionVhsCatalog;
    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Comedy")
    private MovieCatalog comedyVhsCatalog;
    @Autowired
    @MovieQualifier(format=Format.DVD, genre="Action")
    private MovieCatalog actionDvdCatalog;
    @Autowired
    @MovieQualifier(format=Format.BLURAY, genre="Comedy")
    private MovieCatalog comedyBluRayCatalog;
    // ...
}
```

最后，bean 的定义应该包含匹配的限定符值。这个示例也展示了 bean 的元属性可能用 于替代 `<qualifier/>` 子元素。如果可用， `<qualifier/>` 和它的属性优先，但是如果目 前没有限定符，自动装配机制就会在 `<meta/>` 标签提供的值上失效，就像下面这个示例中的 最后两个

bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Action"/>
        </qualifier>
        <!– 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Comedy"/>
        </qualifier>
        <!– 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="DVD"/>
        <meta key="genre" value="Action"/>
        <!– 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="BLURAY"/>
        <meta key="genre" value="Comedy"/>
        <!– 注入这个bean所需要的任何依赖 -->
    </bean>
</beans>
```

4.9.4 CustomAutowireConfigurer

CustomAutowireConfigurer 是 BeanFactoryPostProcessor 的一种， 它使得 你可以注册你自己定义的限定符注解类型，即便它们没有使用 Spring 的@Qualifier 注解 也是可以的。

```
<bean id="customAutowireConfigurer" class="org.springframework.beans.factory.annotation.C
    towireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>example.CustomQualifier</value>
        </set>
    </property>
</bean>
```

AutowireCandidateResolver 的特别实现类对基于 Java 版本的应用程序上下文激 活。在 Java 5 之前的版本中，限定符注解是不支持的，因此自动装配候选者仅由每个 bean 定义中的 autowire-candidate 值来决 定，还有 在 `<beans/>` 元 素中的任 何可用的 default-autowire-candidates 模 式 也 是 可 以 的 。 在 Java 5 之 后 的 版 本 中 ， @Qualifier 注解的存在还有任意使用 CustomAutowireConfigurer 注册的自定义注解也将发挥作用。

不管 Java 的版本，当多个 bean 作为自动装配的候选者，决定“主要”候选者的方式也 是相同的：如果候选者中一个 bean 的定义有 primary 属性精确地设置为 true，那么它就会被选择。

4.9.5 @Resource

Spring 也支持使用 JSR 250 的@Resource 注解在字段或 bean 属性的 setter 方法上的注入。这在 Java EE 5 和 6 中是一个通用的模式，比如在 JSF 1.2 中管理的 bean 或 JAX-WS 2.0 端点。Spring 也为其所管理的对象支持这种模式。

@Resource 使用名称属性，默认情况下 Spring 解释这个值作为要注入的 bean 的名称。

换句话说，如果遵循 by-name 语义，正如在这个示例所展示的：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Resource(name="myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

如果没有明确地指定名称，那么默认的名称就从字段名称或 setter 方法中派生出来。以字段为例，它会选用字段名称；以 setter 方法为例，它会选用 bean 的属性名称。所以下面 的示例中有名为“movieFinder”的 bean 通过 setter 方法来注入：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

> 注意
>
> 使用注解提供的名称被感知 CommonAnnotationBeanPostProcessor 的 ApplicationContext 解析成 bean 的名称。名称可以通过 JNDI 方式来解析，只要你明确地配置了 Spring 的 SimpleJndiBeanFactory。然而，还是推荐你基于默认行为并仅使用 Spring 的 JNDI 查找能力来保存间接的水平。

在使用@Resource 并没有 明确指 定名称的 独占情况 下，和 @Autowired 相似，@Resource 发现主要类型匹配，而不是特定名称 bean 并解析了熟知的可解析的依赖关系：BeanFactory， ApplicationContext， ResourceLoader， ApplicationEventPublisher，MessageSource 和接口。

因此，在下面的示例中，customerPreferenceDao 字段首先寻找名为 customerPreferenceDao 的 bean，之后回到匹配 CustomerPreferenceDao 的主类型。“context”字段基于已知的可解析的依赖类型 ApplicationContext 注入。

```
public class MovieRecommender {
    @Resource
    private CustomerPreferenceDao customerPreferenceDao;
    @Resource
    private ApplicationContext context;
    public MovieRecommender() {

    }
    // ...
}
```

4.9.6 @PostConstruct 和@PreDestroy

CommonAnnotationBeanPostProcessor 不但能识别@Resource 注解，而且还能 识别 JSR-250 生命周期注解。在 Spring 2.5 中引入，对这些注解的支持提供了在初始化回调（4.6.1.1 节）和销毁回调（4.6.1.2 节）中的另一种选择。只要 CommonAnnotationBeanPostProcessor 在 Spring 的 ApplicationContext 中注册， 一个携带这些注解之一的方法就同时被调用了，和 Spring 生命周期接口方法或明确地声明回调方法相对应。在下面的示例中，在初始化后缓存会预先填充，在销毁后会清理。

```
public class CachingMovieLister {
    @PostConstruct
    public void populateMovieCache() {
        // 在初始化时填充movie cache...
    }
    @PreDestroy
    public void clearMovieCache() {
        // 在销毁后清理movie cache...
    }
}
```

> 注意
>
> 关于组合多个生命周期机制影响的细节内容，请参考 4.6.1.4 节，“组合生命周期机制”。

4.10 类路径扫描和管理的组件

本章中的大多数示例都使用了 XML 来指定配置元数据在 Spring 的容器中生产每一个 BeanDefinition。之前的章节（4.9 节，“基于注解的容器配置”）表述了如何通过代码级 的注解来提供大量的配置信息。尽管在那些示例中，“基础的”bean 的定义都是在 XML 文 件中来明确定义的，而注解仅仅进行依赖注入。本节来说明另外一种通过扫描类路径的方式 来隐式检测候选组件。候选组件是匹配过滤条件的类库，并有在容器中注册的对应的 bean 的定义。这就可 以不用 XML 来执行 bea n 的注册了 ，那么你就 可以使用注 解（比如 @Component），AspectJ 风格的表达式，或者是你子定义的过滤条件来选择那些类会有在容 器中注册的 bean。

> 注意
>
> 从 Spring 3.0 开始，很多由 Spring JavaConfig 项目提供的特性作为 Spring Framework 核心 的一部分了。这就允许你使用 Java 而不是传统的 XML 文件来定义 bean 了。看一看 @Configuration，@Bean，@Import 和@DependsOn 注解的例子来使用它们的新特性。

4.10.1 @Component 和更多典型注解

在 Spring 2.0 版之后，@Repository 注解是任意满足它的角色或典型（比如熟知的数 据访问对象，DAO）库的类的标记。在这个标记的使用中，就是在 14.2.2 节，“表达式翻译” 中描述的表达式自动翻译。

Spring 2. 5 引 入 了 更 多 的 注 解 ： @Component， @Service 和 @Controller 。

@Component 是对 Spring 任意管理 组件的通用刻 板。@Repository， @Service 和 @Controller 是对更多的特定用例@Component 的专业化，比如，在持久层，服务层和 表现层。因此，你可以使用@Component 注解你的组件类，但是使用@Repository， @Service 或@Controller 注解来替代的话，那么你的类更合适由工具来处理或和不同的方面相关联。比如，这些刻板注解使得理想化的目标称为切入点。而且@Repository， @Service 和 @Controller 也可以在将来 Spring Framework 的发布中携带更多的语义。因此，如果对于服务层，你在@Component 或@Service 中间选择的话，那么@Service 无 疑是更好的选择。相似地，正如上面提到的，在持久层中，@Repository 已经作为自动异 常翻译的表示所被支持

4.10.2 自动检测类和 bean 的注册

Spring 可以自动检测刻板类并在 ApplicationContext 中注册对应的BeanDefinition。比如，下面的两个类就是自动检测的例子：

```
@Service
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
@Repository
public class JpaMovieFinder implements MovieFinder {
    //为了清晰省略了实现
}
```

要自动检测这些类并注册对应的 bean，你需要包含如下的 XML 元素，其中的 base-package 元素通常是这两个类的的父包。（也就是说，你可以指定以逗号分隔的列表来为每个类引入父包。）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2
    <context:component-scan base-package="org.example"/>
</beans>
```

> 注意
>
> 对类路径包的扫描需要存在类路径下对应的目录。当你使用 Ant 来构建 JAR 包时，要保证你没有激活 JAR 目标中的 files-only 开关。

此外，当你使用 component-scan（组件-扫描，译者注）时，AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor 二者都是隐式包含的。这就意味着两个组件被自动检测之后就装配在一起了-而不需要在 XML 中提供其它任何 bean 的配置元数据。

> 注意
>
> 你可以将 annotation-config 属性置为 false 来关闭 AutowiredAnnotationBeanPostProcessor 的 CommonAnnotationBeanPostProcessor 注册。

4.10.3 使用过滤器来自定义扫描

默认情况下，使用@Component，@Repository，@Service，@Controller 注解 或使用了进行自定义的@Component 注解的类本身仅仅检测候选组件。你可以修改并扩展 这 种 行 为，仅仅 应 用自 定 义的 过 滤器 就 可以 了。 在 component-scan 元素中添加 include-filter 或 exclude-filter 子元素就可以了。每个过滤器元素需要 type 和 expression 属性。下面的表格描述了过滤选项。

表 4.5 过滤器类型

| 过滤器类型 | 表达式示例 | 描述 | | annotation（注解） | org.example.SomeAnnotation | 在目标组件的类型层表示的注解 | | assignable（分配） | org.example.SomeClass | 目标组件分配去（扩展/实现）的类（接口） | | aspectj | org.example..*Service+* | *AspectJ* 类型表达式来匹配目标组件 | | *regex*（正则表达式） | *org.example.Default.* | 正则表达式来匹配目标组件类的名称 | | custom（自定义） | org.example.MyTypeFilter | 自 定 义 org.springframework.core.type.TypeFilter 接口的实现类 |

下面的示例代码展示了 XML 配置忽略所有@Repository 注解并使用“sub”库来替代。

```
<beans>
    <context:component-scan base-package="org.example">
    <context:include-filter type="regex" expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository
        "/>
    </context:component-scan>
</beans>
```

> 注意
>
> 你可以使用`<component-scan/>`元素中的 use-default-filter="false"属性来关闭默认的过滤器。这会导致关闭使用@Component，@Repository，@Service 或@Controller 注解 的类的自动检测。

4.10.4 使用组件定义 bean 的元数据

Spring 组件可以为容器提供 bea n 定义的元数据。你可以使用用于定义 bea n 元数据的@Configuration 注解的类的@Bean 注解。这里有一个示例：

```
@Component
public class FactoryMethodComponent {
    @Bean @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }
    public void doWork() {
        //忽略组件方法的实现
    }
}
```

这个类在 Spring 的组件中有包含在它的 doWork()方法中的特定应用代码。它也提供 了 bean 的定义并且有工厂方法来指向 publicInstance()方法。@Bean 注解标识了工厂 方法和其它 bean 定义的属性，比如通过@Qualifier 注解表示的限定符。其它方法级的注 解可以用于特定的是@Scope，@Lazy 和自定义限定符注解。自动装配字段和方法也是支持 的，这在之前讨论过，而且还有对自动装配@Bean 方法的支持：

```
@Component
public class FactoryMethodComponent {
    private static int i;
    @Bean @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }
    // 使用自定义标识符并自动装配方法参数
    @Bean
    protected TestBean protectedInstance(@Qualifier( "public") TestBean spouse,@Value("#{
        TestBean tb = new TestBean("protectedInstance", 1); tb.setSpouse(tb);
        tb.setCountry(country);
        return tb;
    }
    @Bean @Scope(BeanDefinition.SCOPE_SINGLETON)
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }
    @Bean @Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }
}
```

这个示例为另外一个名为 privateInstance 的 bea n 的 Age 属性自动装配了 String 方法参数 country。Spring 的表达式语言元素通过 `#{<expression>}` 表示定义了属性的 值。对于@Value 注解，当解析表达式文本时，表达式解析器会预先配置来查看 bean 的名 称。

Spring 组件中的@Bean 方法会被不同方式来处理，而不会像 Spring 中@Configuration 的类那样。不同的是@Component 类没有使用 CGLIB 来加强并拦截字段和方法的调用。CGLIB 代理是调用@Configuration 类和创建 bean 元数据引用协作对象的@Bean 方法调用的手 段。方法没有使用通常的 Java 语义来调用。相比之下，使用@Component 类@Bean 方法来 调用方法或字段有标准的 Java 语义。

4.10.5 命名自动检测组件

当组件被自动检测作为扫描进程的一部分时，它的 bean 名称是由 BeanNameGenerator 策略来生成并告知扫描器的。默认情况下， Spring 的刻板注解（@Component，@Repository，@Service 和@Controller）包含 name 值从而提供对应 bean 定义的名称。 注意 JSR 330 的@Named 注解可以被用于检测组件和为它们提供名称的手段。如果在类路径 中有 JSR 330 的 JAR 包的话，这种行为会被自动开启。

如果注解包含 name 值或者对于其它任意被检测的组件（比如那些被自定义过滤器发现的），默认的 bean 的名称生成器返回未大写的非限定符类名。比如，如果下面的两个组件被检测到了，那么名称可能是 myMovieLister 和 movieFinderImpl：

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

注意

如果你不想使用默认的 bean 命名策略，你可以提供自定义的 bean 命名策略。首先，实现 BeanNameGenerator 接口，要保证包含默认的没有参数的构造方法。之后，在配置 扫描器时，要提供类的完全限定名。

```
<beans>
    <context:component-scan base-package="org.example" name-generator="org.example.MyName
</beans>
```

作为通用的规则，要考虑使用注解指定名称时，其它组件可能会有对它的明确的引用。 另一方面，当容器负 责装配时，自动生成名称是可行的。

4.10.6 为自动检测组件提供范围

一般情况下，Spring 管理的组件，自动检测组件的默认和最多使用的范围是单例范围。 然而，有事你需要其它范围，Spring 2.5 提供了一个新的@Scope 注解。仅仅使用注解提供 范围的名称：

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

注意

为范围决策提供自定义策略而不是基于注解的方式，实现 ScopeMetadataResolver 接口，并保证包含了默认的无参构造方法。之后，当配置扫描器时，要提供类的完全限定名：

```
<beans>
    <context:component-scan base-package="org.example" scope-resolver="org.example.MyScop
</beans>
```

当使用特定反而非单例范围时，它可能必须要为有范围的对象生成代理。这个原因在 4.5.4.5 节，“各种范围的 bean 作为依赖”中描述过了。出于这样的目的，在 component-scan 元素中可以使用 scoped-proxy 属性。三种可能的值是：no（无），interface（接口）和 targetClass（目标类）。比如，下面的配置就会启动标准的 JDK 动态代理：

```
<beans>
    <context:component-scan base-package="org.example" scoped-proxy="interfaces" />
</beans>
```

4.10.7 使用注解提供限定符元数据

@Qualifier 注解在 4.9.3 节，“使用限定符来微调基于注解的自动装配”中讨论过了。 那部分中的示例说明了@Qualifier 注解的使用和当你需要处理自动装配候选者时，自定 义限定符注解来提供微调控制。因为那些示例是基于 XML 的 bean 定义的，限定符元数据在 候选者 bean 定义中提供，并使用了 XML 中的 bean 元素的 qualifier 和 meta 子元素。 当对自动检测组件使用基于类路径扫描时，你可以在候选者类中使用类型-级的注解提供限 定符元数据。下面的三个示例就展示了这个技术：

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

> 注意
>
> 对于大多数基于注解的方式，要记得注解元数据会绑定到类定义本身中去，而使用 XML 就允许对多个相同类型的 bean 在它们的限定符元数据中提供变化，因为那些元数据是对于 每个实例而不是每个类提供的。

4.11 使用 JSR 330 标准注解

从 Spring 3.0 开始，Spring 提供了对 JSR-330 标准注解（依赖注入）的支持。这些注解可以和 Spring 注解以相同方式被扫描到。你仅仅需要在类路径下添加相关的 jar 包即可。

> 注意
>
> 如果你使用 Maven，那么在标准 Maven 仓库（http://repo1.maven.org/maven2/javax/inject/javax.inject/1/）中 javax.inject 的 artifact 是可用的。你仅仅需要在 pom.xml 中添加如下的依赖即可：

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

4.11.1 使用@Inject 和@Named 进行依赖注入

除了@Autowired，javax.inject.Inject 还可以是下面这样：

```
import javax.inject.Inject;
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

至于@Autowired，可以在类级别，字段级别，方法级别和构造方法参数级别使用 @Inject。如果你想对应该被注入的依赖使用限定符名称，你应该按如下方式使用@Named注解：

```
import javax.inject.Inject; import javax.inject.Named; public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

4.11.2 @Named：@Component 注解的标准等同

除了@Component，javax.inject.Named 还可以是下面这样：

```
import javax.inject.Inject;
import javax.inject.Named;
@Named("movieListener")
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

使用@Component 而不指定组件的名称是很常用的方式。@Named 可以被用于相同的 情况：

```
import javax.inject.Inject;
import javax.inject.Named;
@Named
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

当使用@Named，当使用 Spring 注解时，可以以相同的方式使用组件扫描：

```
<beans>
    <context:component-scan base-package="org.example"/>
</beans>
```

4.11.3 标准方法的限制

当使用标准注解时，了解一些不可用，但是很重要的特性是必要的，在下面的表格中给 出：

表 4.6 Spring 注解和标准注解的对比

Spring	javax.inject.*	javax.inject 限制/注释
@Autowired	@Inject	@Inject 没有'required'属性
@Component	@Named	
@Scope("singleton")	@Singleton	jsr-330 默认范围和 Spring 的 prototype 相似。但是，要保持和 Spring 一般默认值一致，在 Spring 容器中 jsr-330 的 bean 声 明默认是 singleton 的。要使用另外的 范围，你应该使用 Spring 的@Scope 注解。javax.inject 也提供@Scope 注解。不过这仅 仅用于创建你自己的注解。
@Qualifier	@Named	
@Value	-	不等同
@Required	-	不等同
@Lazy	-	不等同

4.12 基于 Java 的容器配置

4.12.1 基本概念：@Configuration 和@Bean

Spring 中新的 Java 配置支持的核心就是@Configuration 注解的类。这些类主要包括 @Bean 注解的方法来为 Spring 的 IoC 容器管理的对象定义实例，配置和初始化逻辑。

使用@Configuration 来注解类表示类可以被 Spring 的 IoC 容器所使用，作为 bean 定义的资源。最简单的@Configuration 类可以是这样的：

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

这和 Spring 的 XML 文件中的 `<beans/>` 非常类似，上面的 AppConfig 类和下面的代码是等同的：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

正如你看到的，@Bean 注解扮演了和 `<bean/>` 元素相同的角色。@Bean 注解会在下面 的章节中详细地讨论。首先，我们要来看看使用基于 Java 的配置创建 Spring 容器的各种方 式。

4.12.2 使用 AnnotationConfigApplicationContext 实例化 Spring 容器

下面的章节说明了 Spring 的 AnnotationConfigApplicationContext，在 Spring 3.0 中 是 新加 入 的 。这 个 全 能的 ApplicationContext 实 现 类 可 以 接 受不 仅 仅 是 @Configuration 类 作为输入，也可以是普通的@Component 类，还有使用 JSR-330 元数 据注解的类。

当@Configuration 类作为输入时，@Configuration 类本身作为 bean 被注册了， 并且类内所有声明的@Bean 方法也被作为 bean 注册了。

当@Component 和 JSR-330 类 作为输 入时， 它们 被注册 为 bea n，并 且被 假设如 @Autowired 或@Inject 的 DI 元数据在类中需要的地方使用。

4.12.2.1 简单构造

当实例化 ClassPathXmlApplicationContext 时，以大致相同的方式，当实例化 AnnotationConfigApplicationContext 时，@Configuration 类可能被作为输入。这 就允许在 Spring 容器中完全可以不使用 XML：

```
public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(AppConfig. class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

正如上面所提到的，AnnotationConfigApplicationContext 不仅仅局限于和 @Configuration 类合作。任意@Component 或 JSR-330 注解的类都可以作为构造方法的输入。比如：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(MyServiceImpl.class,
    Dependency1.class, Dependency2.MyService myService =
    ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面假设 MyServiceImpl，Dependency1 和 Dependency2 使用了 Spring 依赖注 入注解，比如@Autowired。

4.12.2.2 使用 `register(Class<?>...)` 来编程构建容器

AnnotationConfigApplicationContext 可以使用无参构造方法来实例化，之后使 用 register() 方 法 来 配 置 。 这 个 方 法 当 编 程 构 建 AnnotationConfigApplicationContext 时尤其有用。

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
        AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

4.12.2.3 使用 scan(String..)开启组件扫描

有经验的 Spring 用户肯定会熟悉下面这个 Spring 的 context:命名空间中的常用 XML 声明

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

在上面的示例中，com.acme 包就会被扫描，去查找任意@Component 注解的类，那些 类就会被注册为 Spring 容器中的 bean。AnnotationConfigApplicationContext 暴露 出 scan(String ...)方法，允许相同的组件扫描功能：

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
    AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

> 注意
>
> 记得@Configuration 类是使用@Component 进行元数据注释的，所以它们是组件扫 描的候选者！在上面的示例中，假设 AppConfig 是声明在 com.acme 包（或是其中的子包）中的，那么会在调用 scan()方法时被找到，在调用 refresh()方法时，所有它的 @Bean 方法就会被处理并注册为容器中的 bean。

4.12.2.4 支 持 Web 应 用 的 AnnotationConfigWebApplicationContext

WebApplicationContext 是 AnnotationConfigApplicationContext 的变种，

适用于 AnnotationConfigWebApplicationContext。当配置 Spring 的 Servlet 监听器 ContextLoaderListener，Spring MVC 的 DispatcherServlet 等时，这个实现类就 可能被用到了。下面的代码是在 web.xml 中的片段，配置了典型的 Spring MVC 的 Web 应用 程序。注意 contextClass 上下文参数和初始化参数的使用：

```
<web-app>
    <!-- 配置ContextLoaderListener使用
    AnnotationConfigWebApplicationContext来代替默认的 XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value> org.springframework.web.context.support.AnnotationCon
        figWebApplicationContext
        </param-value>
    </context-param>
    <!-- 配置位置必须包含一个或多个逗号或空格分隔的完全限定
    @Configuration类。完全限定包也可以指定于组件扫描 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>
    <!-- 普通方式启动根应用上下文的ContextLoaderListener -->
    <listener>
        <listener-class> org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 普通方式声明 Spring MVC 的 DispatcherServlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
            <servlet-class> org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <!-- 配置DispatcherServlet使用
        AnnotationConfigWebApplicationContext来代替默认的 XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value> org.springframework.web.context.support.Annotation
                ConfigWebApplicationContext
            </param-value>
        </init-param>
        <!--配置位置必须包含一个或多个逗号或空格分隔的完全限定
        @Configuration类-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>
    <!-- 映射所有的请求到/main/*中来派发servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>
</web-app>
```

4.12.3 构成基于 Java 的配置

4.12.3.1 使用@Import 注解

就像 Spring 的 XML 文件中使用的 `<import/>` 元素帮助模块化配置一样，@Import 注 解允许从其它配置类中加载@Bean 的配置：

```
@Configuration
public class ConfigA {
    public @Bean A a() { return new A(); }
}
@Configuration
@Import(ConfigA.class)
public class ConfigB {
    public @Bean B b() { return new B(); }
}
```

现在，当实例化上下文时，不需要指定 ConfigA.class 和 ConfigB.class 了，仅 仅 ConfigB 需要被显式提供：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(ConfigB.class);
    // 现在bean A 和bean B都会是可用的...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

这种方式简化了容器的实例化，仅仅是一个类需要被处理，而不是需要开发人员在构造 时记住很多大量的@Configuration 类。

在引入的@Bean 定义中注入依赖 上面的示例是可行的，但是很简单。在很多实际场景中，bean 会有依赖其它配置的类的依赖。当使用 XML 时，这本身不是什么问题，因为没有调用编译器，而且我们可以仅仅 声明 ref="someBean" 并且相信 Spring 在 容 器 初 始化 时 可以完 成。 当 然 ， 当 使 用 @Configuration 的类时，Java 编译器在配置模型上放置约束，对其它 bean 的引用必须 是符合 Java 语法的。

幸运的是，解决这个问题分层简单。记得@Configuration 类最终是容器中的 bean- 这就是说它们可以像其它 bean 那样利用@Autowired 注入元数据！ 我们来看一个更加真实的语义，有几个@Configuration 类，每个都依赖声明在其它类中的 bean：

```
@Configuration
public class ServiceConfig {
    private @Autowired AccountRepository accountRepository;
    public @Bean TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {
    public @Bean DataSource dataSource() { /* 返回新的数据源 */ }
}
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(SystemTestConfig.class);
    // 所有的装配都使用配置的类...
    TransferService transferService = ctx.getBean(TransferService.class); transferService
}
```

完全限定引入的 bean 便于导航

在上面的场景中，使用@Autowired 工作正常，提供所需的模块化，但是准确地决定 在哪儿声明自动装配的 bean 还是有些含糊。比如，作为开发者来看待 ServiceConfig， 你如何准确知道@Autowired AccountRepository 在哪里声明的？它没有显式地出现 在代码中，这可能很不错。要记得 SpringSource Tool Suite 提供工具可以生成展示所有对象是 如何装配起来的图片-那可能就是你所需要的。而且，你的 Java IDE 可以很容器发现所有的声明，还有使用的 AccountRepository 类型，也会很快地给你展示出@Bean 方法的位置 和返回的类型。

在这种歧义不被接受和你想有直接从 IDE 中从一个@Configuration 类到另一个导航 的情景中，要考虑自动装配配置类的本身：

```
@Configuration
public class ServiceConfig {
    private @Autowired RepositoryConfig repositoryConfig;
    public @Bean TransferService transferService() {
        // 通过配置类到@Bean方法的导航!
        return new
            TransferServiceImpl(repositoryConfig.accountRepository());
    }
}
```

在上面的情形中，定义 AccountRepository 是完全明确的。而 ServiceConfig 却紧紧耦合到 RepositoryConfig 中了；这就需要我们来权衡了。这种紧耦合可以使用 基于接口或抽象基类的@Configuration 类来减轻。考虑下面的代码：

```
@Configuration
public class ServiceConfig {
private @Autowired RepositoryConfig repositoryConfig;
public @Bean TransferService transferService() {
    return new
        TransferServiceImpl(repositoryConfig.accountRepository());
}
}
@Configuration
public interface RepositoryConfig {
    @Bean AccountRepository accountRepository();
}
@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig
{
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }
}
@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class})
// 导入具体的配置!
public class SystemTestConfig {
    public @Bean DataSource dataSource() { /* 返回数据源 */ }
}
public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(SystemTestConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

现在 ServiceConfig 和 DefaultRepositoryConfig 的耦合就比较松了，并且内 建的 IDE 工具也一直有效：对于开发人员来说会更加简单地获取 RepositoryConfig 实现 类的类型层次。以这种方式，导航@Configuration 类和它们的依赖就和普通的基于接口 代码的导航过程没有任何区别了。

4.12.3.2 结合 Java 和 XML 配置

Spring 的@Configuration 类并不是完全 100%地支持 Spring XML 替换的。一些基本特 性，比如 Spring XML 的命名空间会保持在一个理想的方式下去配置容器。在 XML 便于使用 或是必须要使用的情况下，你也有另外一个选择：以"XML 为中心"的方式来实例化容器， 比如，ClassPathXmlApplicationContext，或者以" Java 为中心"的方式，使用 AnnotationConfigurationApplicationContext 和@ImportResource 注解来引 入需要的 XML。

以"XML 为中心"使用@Configuration 类从一种特定的方式的包含@Configuration 类的 XML 文件启动 Spring 容器是不错的。 比如，在使用了 Spring XML 的大型的代码库中，根据需要并从已有的 XML 文件中创建 @Configuration 类是很简单的。在下面，你会发现在这种"XML 为中心"情形下，使用 @Configuration 类的选择。

以普通的 Spring `<bean/>` 元素声明@Configuration 类

要记得@Configuration 的类最终仅仅是容器中的 bean。在这个示例中，我们创建了 名为 AppConfig 的@Configuration 类，并且将它包含在 system-test-config.xml 文件中作为 `<bean/>` 的定义。因为开启了 `<context:annotation-config/>` 配置，容器 会识别 @Configuration 注解，以合适的方式处理声明在 AppConfig 中@Bean 方法。

```
@Configuration
public class AppConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
    public @Bean TransferService transferService() {
        return new TransferService(accountRepository());
    }
}

system-test-config.xml
<beans>
    <!-- 开启处理注解功能，比如@Autowired和@Configuration -->
    <context:annotation-config/>
    <context:property-placeholder
        location="classpath:/com/acme/jdbc.properties"/>
    <bean class="com.acme.AppConfig"/>
    <bean class="org.springframework.jdbc.datasource.DriverManagerData Source">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>

jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=

public static void main(String[] args) {
    ApplicationContext ctx
        = new ClassPathXmlApplicationContext("classpath:/com/acme/system-t est-config.xml
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

> 注意
>
> 在上面的 system-test-config.xml 文件中，AppConfig 的 `<bean/>` 定义没有声明 id 元素。这么做也是可以接受的，就不必让其它 bean 去引用它了。同时也就不可能从 容器中通过明确的名称来获取它了。同样地，DataSource bean，通过类型自动装配，那么明确的 bean id 就不严格要求了。

使用 `<context:component-scan/>` 来检索@Configuration 类

因为@Configuration 是使用@Component 来元数据注解的，被@Configuration 注解的类是自动作为组件扫描的候选者的。使用上面相同的语义，我们可以重新来定义 system-test-config.xml 文件来利用组件扫描的优点。注意这种情况下，我们不需要 明确地声明 `<context:annotation-config/>` ，因为 `<context:component-scan/>` 开启了相同的功能。

```
system-test-config.xml
<beans>
    <!-- 选择并注册AppConfig作为bean -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder
        location="classpath:/com/acme/jdbc.properties"/>
    <bean class="org.springframework.jdbc.datasource.DriverManagerData Source">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

使用了@ImportResource 导入 XML 的@Configuration 类为中心在@Configuration 类作为配置容器主要机制的应用程序中，使用一些 XML 还是必要的。 在这些情况中，仅仅使用@ImportResource 来定义 XML 就可以了。这么来做就实现了“Java 为中心”的方式来配置容器并保持 XML 在最低限度。

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {
    private @Value("${jdbc.url}") String url;
    private @Value("${jdbc.username}") String username;
    private @Value("${jdbc.password}") String password;
    public @Bean DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}

properties-config.xml
<beans>
    <context:property-placeholder
    location="classpath:/com/acme/jdbc.properties"/>
</beans>

jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=

public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(AppConfig. class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```